NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEFENSE IN DEPTH ADDED TO MALICIOUS ACTIVITIES SIMULATION TOOLS (MAST)

by

Adam M. Farber
Robert A. Rawls

September 2015

Thesis Advisor:	Gurminder Singh
Co-Advisor:	John Gibson

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2015	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE DEFENSE IN DEPTH ADDED TO MALICIOUS ACTIVITIES SIMULATION TOOLS (MAST)			**5. FUNDING NUMBERS**
6. AUTHOR(S) Farber, Adam M. and Rawls, Robert A.			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

With its ever-increasing reliance upon computers and networks in all facets of operation and administration, the U.S. military is becoming increasingly vulnerable to computer and network-based threats. Military technicians' ability to prevent and mitigate these threats is a skill that must be learned and practiced; for this reason, the Malicious Activities Simulation Tool (MAST) was created.

The ongoing training required to defend networks and ensure DOD network policies are implemented correctly is costly and time consuming. A solution was needed to facilitate training system operators and administrators in potentially inauspicious environments, and to be adaptable to emerging threats. Since the proposed solution is on local systems with communication traveling over untrusted networks, a defense in depth plan ensures no undue consequences occur during MAST use.

14. SUBJECT TERMS MAST, SSL / TLS, digital signature, Malicious Activity Simulation Tool, defense in depth			**15. NUMBER OF PAGES** 107
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

DEFENSE IN DEPTH ADDED TO MALICIOUS ACTIVITIES SIMULATION TOOLS (MAST)

Adam M. Farber
Lieutenant, United States Navy
B.S., United States Naval Academy, 2004

Robert A. Rawls
Lieutenant, United States Navy
B.S., University of Virginia, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2015

Authors:	Adam M. Farber Robert A. Rawls
Approved by:	Gurminder Singh, Ph.D. Thesis Advisor
	John Gibson Co-Advisor
	Peter Denning, Ph.D. Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

With its ever-increasing reliance upon computers and networks in all facets of operation and administration, the U.S. military is becoming increasingly vulnerable to computer and network-based threats. Military technicians' ability to prevent and mitigate these threats is a skill that must be learned and practiced; for this reason, the Malicious Activities Simulation Tool (MAST) was created.

The ongoing training required to defend networks and ensure DOD network policies are implemented correctly is costly and time consuming. A solution was needed to facilitate training system operators and administrators in potentially inauspicious environments, and to be adaptable to emerging threats. Since the proposed solution is on local systems with communication traveling over untrusted networks, a defense in depth plan ensures no undue consequences occur during MAST use.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3DES	Triple Data Encryption Standard
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption System
CBC	Cipher Block Chaining
DES	Data Encryption Standard
DOD	Department of Defense
DSA	Digital Signature Algorithm
ECB	Electronic Cookbook
ECDSA	Elliptic Curve Digital Signature Algorithm
EN	End Node
MAC	Message Authentication Code
MAST	Malicious Activity Simulation Tool
OOP	Object Orientated Programming
RMF	Risk Management Framework
RSA	Rivest, Shamir, Adleman
SE	Scenario Execution Server
SG	Scenario Generation Server
XOR	Exclusive Or

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like to thank Dr. Gurminder Singh and Mr. John Gibson for their mentorship and time during our research. Their direction, guidance, and expertise were instrumental to the successful completion of this thesis. They motivated us during the challenging portions of our research and challenged us to learn, dig deeper, and push our own limits throughout the process.

From Adam: I would like to thank my family for their unwavering faith in me and the support they have shown throughout this entire endeavor. Without them, I would have been a ship without a rudder and not been able to finish. Rebecca has helped me with more than she will ever know and has provided the backbone to the family while I was locked in the "dungeon" working on this "book," as the kids call it. Thank you from the bottom of my heart for all that you do. Also, thank you to my thesis partner Alex for putting up with me throughout this process and dealing with all my little idiosyncrasies; we did it!

From Alex: First and foremost, I would like to thank God for His continued blessing and for giving me the perseverance to complete my degree. Secondly, Adam, thank you for putting up with me throughout these trials—I chose this thesis for the thesis partner, not the subject, and I chose well. Thanks for leading the charge on the writing. And finally, I would like to sincerely thank my wonderfully supportive wife, Heather, who consistently contributes more to the family than I could ever ask. All of my success can be attributed to her selfless devotion to me and our growing family. She is my best friend and my inspiration to excel. I could not have done it without her.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

With its ever-increasing reliance upon computers and networks in all facets of operation and administration, the Department of Defense (DOD) is becoming increasingly vulnerable to computer and network-based threats. Network operators and administrators' ability to prevent and mitigate these threats is a skill that must be learned and practiced; for this reason, the Malicious Activities Simulation Tool (MAST) was created.

The ongoing training required to defend networks and ensure DOD network policies are implemented correctly is costly and time consuming. A solution is needed to facilitate training technicians and administrators in potentially inauspicious environments, and to be adaptable to emerging threats.

Autonomous training and evaluating is needed due to a deficit in evaluation of personnel and resources in general. The training required to maintain a level of proficiency commensurate with the capabilities of current and potential adversaries is more than the Navy and the DOD can afford to allocate in a time of significant fiscal constraints. The need is recognized for an ability to conduct economical and timely on-the-job training. Thus, MAST is created to provide the opportunity for services to train on the very networks they are responsible for operating [1].

Currently, MAST uses a tiered approach to operations. There is a master server, called the MAST Scenario Generation Server (SG), where training scenarios are created, signed, and distributed.

The next layer of MAST is the Scenario Execution Server (SE); this is the focal point for many functions of the overall MAST architecture and contains the majority of MAST's complexity and code. The SE communicates directly with the SG as well as the End Nodes (EN) for a particular scenario. It is the server that executes a given scenario and, as such, is responsible for all command and control functions of that scenario.

The final layer of MAST is the EN; this is the service running on each participating client computer on the network being evaluated or for which the administrators are being trained. The MAST EN is the program that enables the SE to interact with the end computer and user during the scenario. Additionally, it provides relevant information back to the SE.

MAST has been in a continual state of development since its inception almost four years ago. Much work has been done on the MAST prototype to enable it to participate in real-world exercises as a proof of concept. As MAST matures and moves towards becoming possibly a program of record, there must be a concerted effort to document improvements and progress toward final acceptance as a program of record.

MAST was evaluated using the DOD Risk Management Framework (RMF) by Brian Diana in 2015[1] and was found substantially lacking in several critical areas. These problems are examined in detail in Chapter III.

A. THESIS OBJECTIVE

MAST currently communicates in the clear, potentially over untrusted networks, and lacks validation of where, or from whom, any communication originates. This issue was addressed in Diana's thesis [1]. This thesis increases MAST's level of security and assurance by addressing several of the problems identified by Diana without sacrificing functionality or ease of use.

The focus of this thesis is the correction of security issues identified in [1], these issues relate to confidentiality, integrity, availability, non-repudiation and authentication. As an academic prototype rather than a product developmental program, MAST was inadequately prepared for the DOD RMF; nevertheless, it was assessed for viability as a program to be offered to the DOD. This thesis evaluates methods to correct seven of the security deficiencies identified in [1], mainly focusing on confidentiality, integrity, availability, non-repudiation, and authentication. Furthermore, it implements the proposed solutions to the greatest extent possible to bring MAST closer to becoming a viable option for the DOD to offer on-the-job training for system administrators and commanders of forces in both the cyber realm and the operational environment.

B. METHODOLOGY

There have been many improvements to security on the Internet in recent years. Many libraries have been created to facilitate the implementation of secure communications over insecure channels. One such method is the use of digital signatures whenever sending information between two end-points that might have to traverse untrusted network paths. Another method is to utilize a secure application-level connection, such as the Secure Socket Layer (SSL) protocols, which have progressed to Transport Layer Security (TLS) protocols. These approaches are not mutually exclusive and provide their own strengths and weaknesses. This thesis employs both methods in concert to provide defense in depth.

The core modules of MAST are implemented in Java and thus allow for the use of the Java Cryptography Architecture (JCA), enabling the use of digital signatures and the creation of TLS/SSL connections. Once these additions have been implemented, they must be thoroughly verified and validated, the details of which are discussed in Chapter IV. Our testing included penetration testing, man in the middle attacks, and unauthorized/malicious transmissions.

C. THESIS OVERVIEW

The thesis is organized into five chapters. The first two chapters provide a background of MAST and encrypted communications. Chapter III evaluates the specific deficiencies of MAST that this thesis aims to correct, the status-quo solutions to such problems, and the chosen implementations. Chapter IV covers the testing of the implementations that are discussed in Chapter III. Finally, Chapter V concludes the thesis with a summary of how well the solutions addressed the deficiencies addressed in Chapter III, along with possible ways to continue to improve MAST.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A lack of secure communication in MAST is an issue that needs to be addressed for MAST to continue developing and re-enter the DOD RMF. Chapter III reviews more in depth of what exactly the secure communication issues are currently in MAST. This chapter provides technical background for the proposed solution presented in Chapter III. Commonly held assumptions about encrypted communication are that there are three main pillars to it: Confidentiality, Integrity, and Availability [2]. Recently there have been two derived additions to these pillars; those are authenticity and non-repudiation. This chapter explores how these three major pillars and two derived pillars interact and ensure that only the intended recipients are the ones to know the content of the communications.

A. CONFIDENTIALITY

Confidentiality is important to encrypted communication because people want to ensure that what is being said between them is not intercepted by anyone. The confidentiality of communication is achieved by encryption. This section discusses the methods for that encryption, either symmetric encryption or asymmetric encryption, and the strengths and limitations of each type of encryption.

Encryption plays a major role in the ability for someone to hide the contents of a message from prying eyes. Throughout the ages encryption has been evolving from the simplest forms to more complex and elaborate. Encryption revolves around the Kerckhoff principle: the encryption scheme must depend only on the secrecy of the key and not on the method of encryption [3]. The two types of encryption are symmetric and asymmetric.

1. Symmetric Encryption

Symmetric encryption is a very secure method of encryption when done correctly but it is not without its shortcomings. The basic necessity of symmetric encryption is that both the sender and receiver have the same key for the decryption. This is both the strength and weakness of symmetric encryption; only those who have the key should be able to decrypt the messages but how does the key get to those people that need it?

(1) Mono-Alphabetic Substitution

One of the earliest forms of encryption dates back to the time of Julius Caesar and is thus named after the famed leader. The Caesar cypher is a substitution encryption where the original letter is replaced, or substituted, with a different letter so that the original word is unrecognizable. The substitution is with another letter of the same alphabet for the Caesar cypher, the alphabet is rotated by a set number of places, N, and then wrapped around. Table 1 shows how this would work using the English alphabet; on top is the normal alphabet and underneath is the shifted alphabet using N = 3, that is, a circular shift of three places.

Table 1. Caesar Cypher

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W

The Caesar cypher is a prime example of a mono-alphabetic cypher and how different variants of the cypher can make it more difficult to decrypt. While there are only 25 different possible shifts that could be used for N in the basic English alphabet, if a person allowed for any permutation of the alphabet for the original substitution, that is not being limited to simple circular rotations, then there are a possibility of over 400,000,000,000,000,000,000,000,000, or 26-factorial, different possible combinations that someone might have to try in order to decode the encrypted message without the proper key [4].

For a time, this was a very successful method of encoding communications; however, eventually an easy way to decode the message without the decoding key was developed. This method used analysis of texts to find the most frequently occurring letters in a language and use that information to decipher the encrypted message. An example of this would be to examine the English language and find that the letter E is the most frequent letter, appearing 12.6 percent of the time [4]. Knowing the frequency of the letters one can find the most frequently appearing letter in the encoded message and start replacing with the original letters. This method of breaking the encryption forced development of a more secure way to encode.

(2) Poly-Alphabetic Substitution

A second form of symmetric encryption is a poly-alphabetic substitution cypher. As the name implies, this new form of substitution cypher uses more than one alphabet to encode the message. An Italian architect, Leon Battista Alberti, first used poly-alphabetic substitution in the middle of the fifteenth century for encryption [4]. However, the first real implementation of this theory of encryption was not used until a French man by the name of Blaise de Vigenère devised a method to use it. The final method that was developed was to use twenty-six different shifts of the alphabet so that a reference table was produced as seen in Table 2.

Table 2. Vigenère Table, from [4]

	a	b	c	D	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
26	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The manner in which this encryption method worked was to choose a key word; for this example the keyword is THESIS and the plaintext message is "what a great work was written." The key word is repeatedly written over the plaintext message until their lengths are the same, then the first letter of the keyword is used to find the line to use for the shifted cipher for that letter. The plaintext letter is found on that line and the letter where the two lines intersect is used as the substitution. This example is seen in Figure 1; by taking the first letter of the plaintext, "w," and intersecting the column with the row from the first letter of the keyword, "T," a "P" is found; this is highlighted in Table 2.

T	H	E	S	I	S	T	H	E	S	I	S	T	H	E	S	I	S	T	H	E	S	I	S
w	h	a	t	a	g	r	e	a	t	w	o	r	k	w	a	s	w	r	i	t	t	e	n
P	O	E	L	I	Y	K	L	E	L	A	G	K	R	A	S	A	O	K	P	X	L	M	F

Figure 1. Vigenère Cipher Example

Figure 1 shows the strength of the poly-alphabetic encryption because the same letter 'w' that appears four different times in the plaintext has a different letter substituted each time. This protects the plaintext from the language analysis that could easily break the mono-alphabetic cipher. However, after taking a closer look, if the encrypted text is long enough then a more complicated analysis can be performed. Too in depth for this thesis is how exactly the analysis is performed; but the encrypted text can be broken down into multiple different mono-alphabetic ciphers that can then be frequency analyzed to break the encryption.

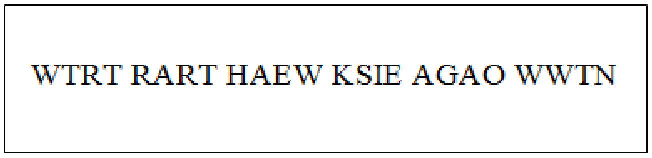
(3) Transposition

Another method of hiding the content of a message is to simply transport some of the letters into different positions with an order to it so that the intended recipient can decrypt it. There are numerous different forms of transposition ciphers to include: rail, route, and columnar or keyed. All of these methods use the movement of the plaintext message to encode the content. The strength of the transposition cipher is that it provides security through obscurity. The message is hidden in plain sight and just looks like garbled text. Anyone trying to decipher the text will think it is either a mono-alphabetic or poly-alphabetic substitution.

An example of the rail transposition is shown in Figure 2. Using the same text as before, the new encryption is seen in Figure 3

```
w     t     r     t     r     a     r     t
   h     a     e     w     k     s     i     e
      a     G     a     o     w     w     t     n
```

Figure 2. Rail Transposition



WTRT RART HAEW KSIE AGAO WWTN

Figure 3. Rail Transposition Encrypted Message

The strength of the transposition cipher is also its greatest weakness. The message is in plain sight and once an analysis realizes that a substitution is not being used, the analysis can turn toward transposition. Focusing on transposition, an analysis can be conducted by simple rearrangement of the letters to find anagrams that sense of the message. The simplest way to avoid the plain sight vulnerability of the transposition is to use a combination of a poly-alphabetic substitution in conjunction with a transposition. This would play to the strength of both forms of encryption and also alleviate most of the weaknesses of both as well. The frequency analysis used against a substitution is made more difficult by the letters being rearranged within the message by the transposition cipher. The combination of the two methods does not mean that the message is perfectly protected; it just means that the message is harder to break.

(4) Block Cypher

As the name implies, a block cypher takes a fixed size of data, a block, and encrypts it to the same size block of encrypted text. A block cypher also requires a key for the encryption and decryption to take place. Typically, the block size for this type of encryption is either 128 bits or 256 bits (16 or 32 bytes/characters), if the data that needs to be encrypted does not fit into a standard block size it will be padded to fit.

Data Encryption Standard (DES)

Data Encryption Standard (DES) was the standard of encryption for a long time because of the method in which it would encrypt the data. There are 16 rounds for the DES system to work; each round will choose a sub-key from the original 56-bit key. DES splits the data into two streams, a left stream and a right stream, each 32 bits in length. The right stream is then expanded to 48-bits to be combined via the exclusive-or (XOR) operation with the 48 bits that were chosen from the 56-bit key. After the XOR operation the stream is put through a substitution table look-up that takes 6-bits at a time and turns it into 4 bits, thus going from 48 bits back to 32 bits. The new 32-bit chunk is then XOR'ed with the left stream 32 bits to become the new right chunk and the old right chunk becomes the new left chunk. This constitutes one round for the cipher; upon completion of all rounds of DES, the 32-bit chunks are then appended together to form the 64-bit block back together. This is illustrated in Figure 4 [5].

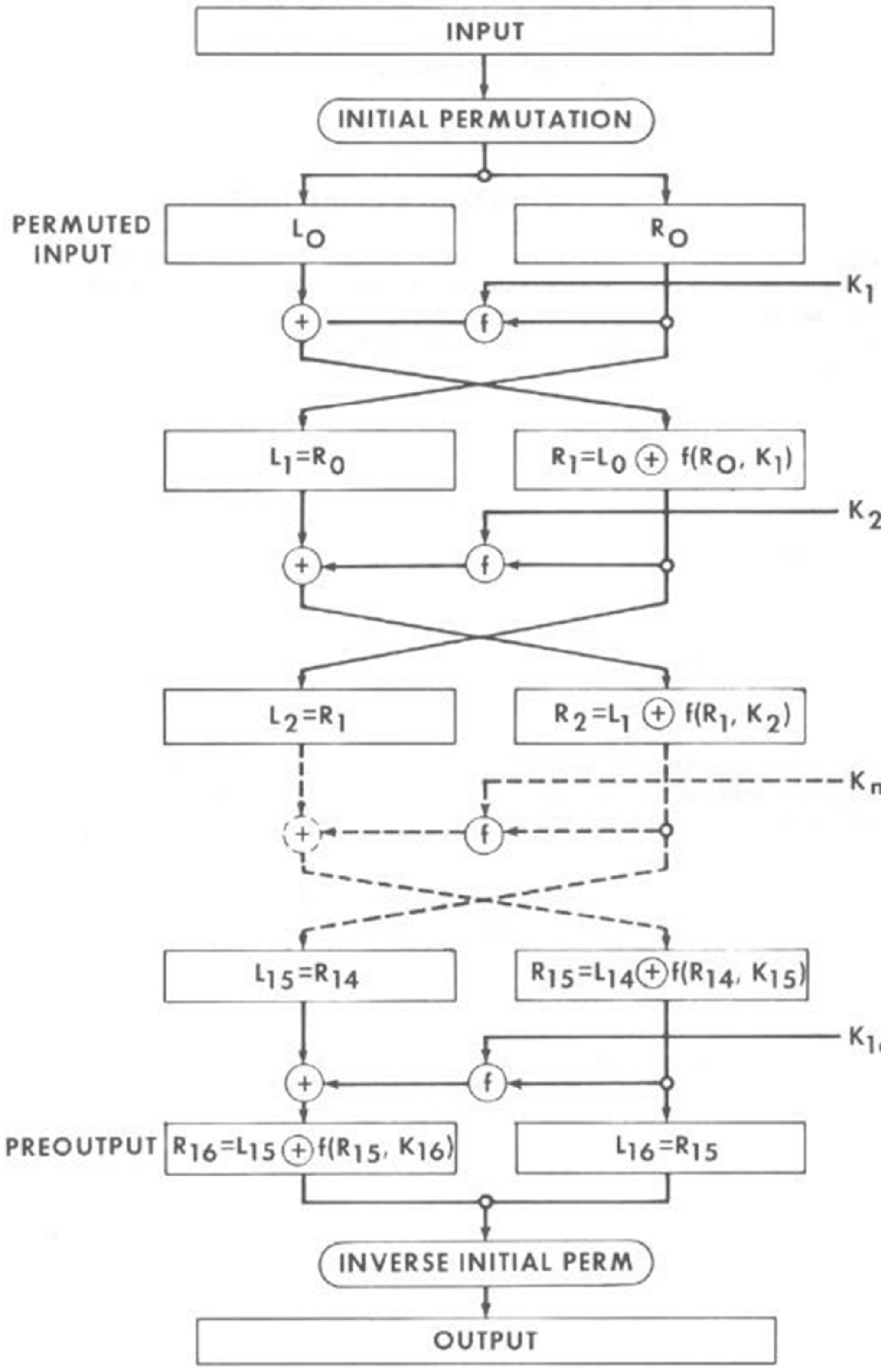


Figure 4. DES Encryption, from [5]

A problem with DES is that it has a small key size. Further, it is a slow process to create the encrypted data; this is because it will only work on a set of 64 bits or 8 bytes of data at a time. Another problem facing DES is the complementation property. As seen in [6], if there is a text, T, such that it was created using DES of a plaintext, P, and a key, K, then:

$$T = DES(P,K)$$

Then also there is a bit-to-bit complement of T:

$$T' = DES(P',K')$$

If the original P is encrypted using all 2^{55} keys K with a least significant bit being 0 then a T'' is created. After each T'' is created, it can be compared with T and T'. If it matches T or T' then the K or K' that were used to generate T and T', respectively, is most likely the key. While this does not arithmetically defeat DES, it does reduce the time taken to discover the key by half, by way of comparing a single T'' to two key values K and K' the work load is reduced.

a. Triple Data Encryption Standard (3DES)

An improvement to DES was to make the process more secure by having the process run three times with either two keys or three keys. This makes the process more secure and harder to break; but it also takes an already slow process and makes it three times slower. Another improvement over DES is the increase of key size for 3DES from a 64-bit key size to yield a 56-bit key to either 128-bits or 192-bits key size to yield two or three, respectively, 56-bit key chunks resulting in a 112-bit or 168-bit key. These improvements have helped strengthen the DES encryption but some of the weaknesses from DES are carried over to 3DES. These weaknesses include the complementary property, as well as if a 0-key, a zero for the entire key, is used for the process. A 0-key would make all of the three keys the same and thus making 3DES the exact same as DES and thereby making no improvement over DES.

b. Advanced Encryption Standard (AES)

The shortcomings of DES and 3DES were enough for a need to have another encryption process created and the National Institute of Standards and Technology (NIST) challenged academia to create something better. There were five finalists and of those five, Rijndael was selected to become AES [3]. The way that AES works is similar to DES with XOR, shifting, and substitution-mapping but different enough that it is better because the bits come out of the substitution mapping and then are shifted before going into the mix column; this helps alleviate the complement problems of DES. The start of AES is similar to DES in that there is substitution of bytes in the chunk. After the substitution takes place the rows of the chunks are shifted and then the subsequent columns are mixed around and finally the key is applied into the chunks. All of this together creates one round of AES. Multiple rounds are needed for AES to be complete. A single round of AES is illustrated in Figure 5. This figure starts at the top with breaking the data into 16 blocks to be fed into the substitution mapping. When the substitution is complete those bits are fed into the mix column. The AES mix column is similar to DES in that the bits are separated and then XOR'ed with other halves of different substitution mappings. The same substitution mapping bits are used for all four rows of the mix column and then finally rejoined together before the key is applied and the bits are then divided back into their original sized chunks.

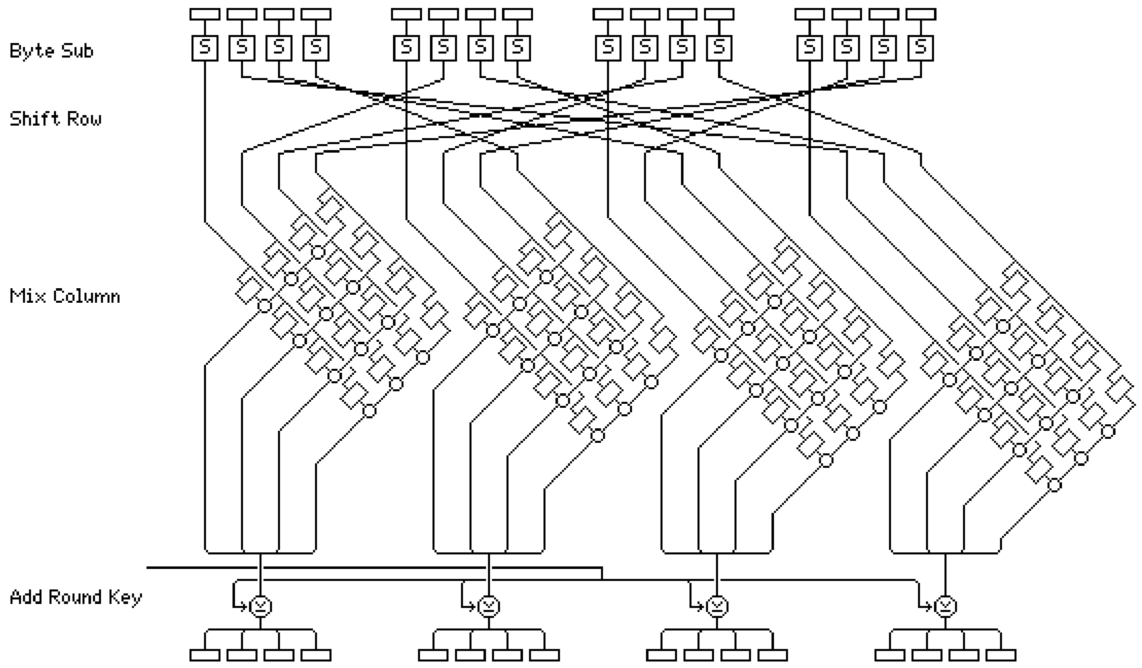


Figure 5. AES Encryption

AES has no known weak keys or semi-weak keys [7]. AES is used throughout the IEEE 802.11 standard as an acceptable method for various encryption requirements as well as it is in use for IPSec [8], [9]. AES strengths will be discussed more in Chapter 3 where the use of AES is shown in the implementation of Secure Socket Layer / Transport Layer Security (SSL/TLS).

(1) Block Ciphering Modes

Since block ciphering takes a fixed block size and data is usually larger than that block size, a method had to be created to take the block size and use it to encrypt the larger amount of data. This section will cover two of the modes that have been used for block ciphers: Electronic Codebook and Cipher Block Chaining; there are more modes but they are beyond the scope of the thesis and these two are the more common modes used [3]. Both of these methods can use DES, 3DES, or AES. For the purposes of this thesis, the discussion will be based on the use of AES.

Electronic Codebook (ECB)

This is the simplest and the weakest mode for block ciphering. For this mode of block ciphering, the data is broken into even sized blocks and encrypted as individual blocks. If the data does not evenly divide into blocks then padding is added to generate the last block for encrypting. Each block is encrypted independent of the other blocks with no influence on any of the other blocks being encrypted. While this makes the encryption faster it does not help if there are large portions of the data that are the same or if data is repeated. This can be seen in Figure 6, an ECB encryption of a picture of the Linux penguin.

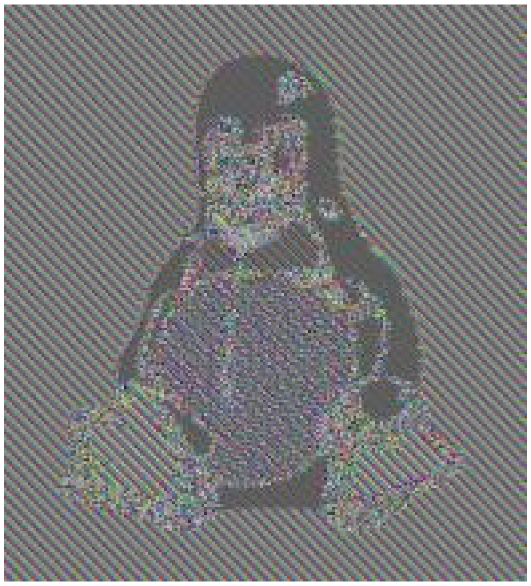


Figure 6. ECB Encryption

As can be seen in Figure 6, the picture can still be discerned. The encryption happened; however, since there was a lot of similar data and there was no change in how that data was encrypted the image changes a little but not completely. This is why ECB is not a very good mode of block ciphering.

Cipher Block Chaining (CBC)

Where ECB had shortcomings and failures, Cipher Block Chaining (CBC) corrected them and is a good block mode to encrypt data. The concept is similar to ECB as the data is broken into blocks and padding is added if needed. The major difference between ECB and CBC is that CBC uses an initialization vector. The initialization vector is XORed with the first block of plaintext; the result is then is put through the block encryption. The output serves as the initialization vector for the next block of ciphering. This is illustrated in Figure 7.

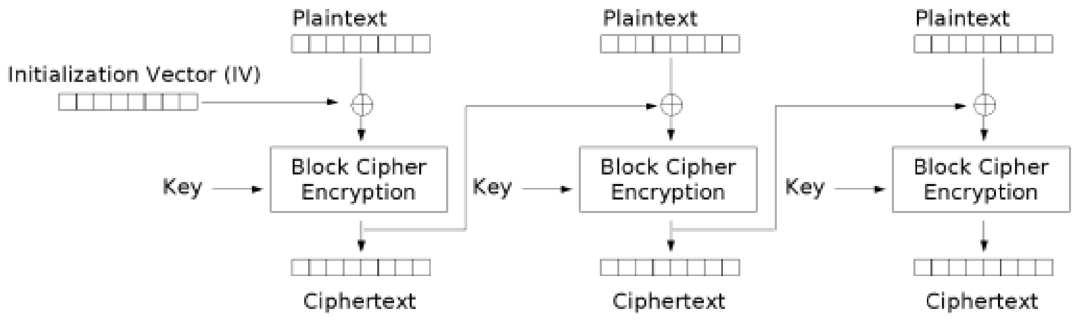


Figure 7. Cipher Block Chaining Mode Encryption

As can be seen in Figure 7, the problem becomes what to use as the initial vector for the first block of code. A few options are available for the first block of code: fixed, counter, random, and nonce-generated. The first option, a fixed initialization vector, is not a good one because similar blocks of data are always encrypted similarly. The second option of using a counter is almost as bad as the fixed option. The reason that the counter option is bad is because much real world data could start similarly and the counter could cancel any differences in the initial XOR causing the blocks to turn out similar enough to be susceptible to attack. The third option is to use a random initial vector. The random initialization vector is usually added to the message as the first cipher block so that the recipient will know the random initialization vector used to encrypt the text. This presents a problem for security because an attacker could intercept the data and try the first block as an initialization vector to decrypt the data thereby enabling a man-in-the-middle attack. While combating the problem associated with ECB and a fixed or counter initialization vector, it tends to add a large amount of data to small data that needs to be encrypted if the original data to be send was small, such as a short message. The final option is to use a nonce, or number used once, as the initialization vector. This nonce is unique in that it is only used once with any key so as to make it more difficult to break by an advisory. When the message is sent the nonce must be encrypted and passed with the data, this does not add too much extra data to the message and is easily accomplished [3].

2. Asymmetric Encryption

While symmetric encryption is a great tool to use for secure communication and data, it does have shortcomings. With symmetric encryption all parties that want to encrypt and decrypt the same message must have the same keys for the process to work correctly. This becomes a problem of scale when a large group of people or machines need the information that is being encrypted. Also a problem of key distribution arises because the keys must be distributed to the people or machines that require them, and doing so securely is difficult.

A different method of providing confidentiality is to use asymmetric encryption. This method uses one key or keys to encrypt information and a different set to decrypt the information. This helps to solve the problem of all parties needing the same key for encryption and decryption. The problem of key compromise is addressed with asymmetric encryption, in that if the receiver's key is compromised than little damage is done to the encrypted message. This will be explained further later in the chapter.

Public key encryption is used synonymously with asymmetric encryption. Keys exist in pairs: a public key and a private key. A public key is a key that is available to anyone and everyone. The private key is beholden to the person for whom it was created to do the encryption or decryption. The two most widely known public key encryption methods are the Diffie-Hellman key exchange and the Rivest, Shamir, and Adleman (RSA) method.

Diffie-Hellman

Diffie-Hellman (D-H) is an easily explained secret key exchange between people. The strength of D-H is brought about through modulo math. There are private numbers that must be kept private and public numbers that must be exchanged between people and a way for all parties to agree on the secret number. D-H uses extremely large prime numbers; but for purposes of this explanation smaller numbers will be used and an assumption of an understanding of modulo math and prime numbers is made.

Suppose two people, Rebecca and Heather, want to exchange secret muffin recipes and to do this they have to agree on a secret number to use for encrypting the recipes. To do so they have to tell each other the public numbers to use. They choose a prime number and a prime root modulo number of the prime number, 23 and 5 respectively. Now each chooses a secret number known only to herself: neither knows the other's secret number. Rebecca chooses her secret number to be 8 and then sends Heather the value, R, which is:

$$R = 5^{8} \bmod 23 = 16$$

Heather also does the same calculation with her secret number, which she chose as 13:

$$H = 5^{13} \bmod 23 = 21$$

Rebecca sent the number 16 to Heather; who in turn sends Rebecca the number 21. With these two numbers, Rebecca and Heather now have the knowledge to access their shared secret number. Rebecca achieves the secret number by calculating it:

$$\text{Secret} = 21^{8} \bmod 23 = 3$$

Heather does the same calculation with the number sent to her from Rebecca.

$$\text{Secret} = 16^{13} \bmod 23 = 3$$

Now Rebecca and Heather have a number that is only known by the two of them. This is asymmetric because anyone can know the prime number and the prime root modulo number; but without knowing the secret number for Rebecca and Heather there is a very small chance that the secret number could be discerned by an attacker.

Rivest, Shamir, and Adleman (RSA)

While the D-H key exchange is good for getting a secret number or key to different people, the data or message being sent is not encrypted. For this purpose a form of encryption must be used and one was created that used asymmetric keys to achieve encryption without the need for everyone to have the same keys. Rivest, Shamir, and Adleman devised a method that would let a person encrypt a message, or data, in a way that it can be sent to the recipient without the need for a key exchange [10]. A recipient would just publish a public key for anyone to use and then retain a private key for the decryption. Similar to the D-H key exchange, there are prime numbers and modulo math used, in conjunction with Euclid's algorithm, and for the following example an understanding of each is assumed. Also like the previous example, smaller prime numbers will be used but the real RSA encryption uses much larger prime numbers ones that are up to 2048 bits in size.

Rebecca and Heather still want to exchange muffin recipes but this time they want to encrypt the recipes and send them in one step instead of the two steps required with the D-H key exchange followed by encrypting.

Rebecca picks two prime numbers, P_1 and P_2, which for this example shall be taken to be 7 and 13, respectively. These values are to remain secret and known only to Rebecca. She multiplies these two numbers together and publishes the product (P), with another number, E, as her public key. The value of E must be such that it is relatively prime to the value $(P_1-1)(P_2-1)$, referred to as the Euler's totient function [11]. These two numbers, P and E, are made known to anyone that wants to send Rebecca a message.

Heather, knowing Rebecca's public key, converts her recipe to a number, for the purposes of this example that number is 76 or M. In order to produce the encrypted text Heather runs her number through the equation:

$$\text{Encrypted Text (C)} = M^E \bmod P$$

This will produce a number, in this example 20, that is then sent to Rebecca. Rebecca then uses her two private numbers and calculates the decryption key, D. This is done by using the modulo multiplicative inverse of E and P such that:

$$E \times D = 1 \bmod (P_1-1)*(P_2-1)$$

The result for this example is that D is equal to 5. Thus, D replaces E in the original equation just like C replaces M. The decryption equation looks similar to the encryption equations but with the substitutions:

$$\text{Decrypted Text (M)} = C^{D} \bmod P$$

After all the substitution and math is worked out, Rebecca would arrive at having the same number for M as Heather had sent [4]. So long as Rebecca does not reveal her secret numbers, anyone can send an encrypted message to Rebecca and she should be the only person who can decrypt it. This is the strength of RSA encryption.

B. INTEGRITY

Integrity is a pillar of secure communication because the people sending data to each other need to be able to rely on the fact that the message is received exactly as it was sent. The easiest way to think about integrity is to remember the childhood game of telephone, where children would sit in a line and at one end the first child would whisper a “secret” sentence into the ear of the child next to him. The sentence would be repeated from one child to the next until the last child in the line would say what she had heard from her neighbor and it would be compared to the original sentence; rarely did the two sentences match. While this is funny in a party environment it takes a turn for the worse if the original message was, “Do not attack at dawn” and the message received was “Attack at dusk” or possibly “Do attack at dawn.” In the following paragraphs threats to integrity and possible protection against these threats will be discussed.

1. Threats to Integrity

Two common forms of threats to integrity exist: manipulation of data and the corruption of data. This section examines these two different threats to integrity and how the two could occur in the real world.

Manipulation

The example just provided, the message “Attack at dusk,” is an example of the message being manipulated while in transit. Manipulation is the willful changing of the contents of the message so that a different result is achieved when the recipient receives the message. The manipulation could be subtle or substantial but if the manipulation is undetected, it is successful.

Corruption

"Do attack at dawn" may be considered an example of simple corruption of the original message, assuming the individual passing it on had simply failed to hear the word "not." This corruption is unintended changing of the message that happens because of transfer methods of the message or data. Corruption could happen because of a power surge along the lines of communication, a lightning strike, a simple data corruption on the storage medium, or wind shifts that affect the swaying of tree branches in a wireless environment. The method of corruption is not important; detecting the corruption, and correcting it when possible, is.

2. Protection of Integrity

While there are intended and unintended forms of integrity violations, there are a few different ways to help protect data from such violations. Two methods reviewed in this thesis are hash functions and parity code. These methods will not indicate what the manipulation is or where the corruption occurred; rather they just help with the ability to detect that there was manipulation or that corruption had happened.

a. Parity Check

A simple error check for integrity is the parity check. This could be either odd parity or even parity, which means that a bit is appended to the set of all bits in the data such that the count of set ("true") bits is either even or odd. An example of this is seen in Table 3.

Table 3. Odd Parity Check

Original Data	Parity Bit	Modified Data	Detection
0 0 0 0 0 0 0 0	1	0 0 0 0 0 0 0 1	Yes
0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 0	Yes
0 0 0 0 0 0 0 0	1	0 0 0 1 1 0 0 0	No
0 0 0 0 0 0 0 0	1	0 0 0 1 1 1 0 0	Yes
0 0 0 0 0 0 0 0	1	0 1 0 1 0 1 0 1	No
0 0 0 0 0 0 0 0	1	1 0 1 0 1 0 1 0	No
0 0 0 0 0 0 0 0	1	1 0 1 0 1 0 0 0	Yes
0 0 0 0 0 0 0 0	1	1 1 1 1 1 1 1 1	No

Table 3 shows an odd parity for an eight-bit data stream. The ninth bit is added so that the sum of the stream is made odd. So long as no even number of bits are corrupted or changed than the parity check does not fail. Various means exist to produce a more complex parity check such that the likelihood of detecting changes in the "checked" data is more probable. Another easy way to check for corruption or manipulation is to use a hash function.

b. Hash Function

A hash function takes any size of data and converts it to a fixed-size amount of data, usually between 128–1024 bits. This means if a hash function is of 128 bits in size it will take any size of data and manipulate it until it is represented by a 128-bit block. In the extreme, if the data is 1 bit long than it will manipulate the data in a manner that it will produce 128 bits for the hash value. Similarly, if the data were a Gigabyte of data the same hash function would manipulate the data in such a way that still only a 128-bit block of data would be produced.

The strength of this technique is that a hash function could produce a random mapping of the original data to the hash value produced. A good hash value will have high collision resistance; that is, it would be highly unlikely that two different blocks of data would create the same hash value for a given function. This random mapping and collision resistance make hashing a good way to help protect the integrity of data; that is a small change, of even one bit, in the original message could cause a large change in the hash value and the change would be detected. The randomness of the mapping ensures that given just a hash value, the original data could not be recovered or reverse engineered.

Two different hash functions are the Secure Hash Algorithm (SHA) and the Message Digest (MD) algorithm. MD5 is the current iteration of the message digest algorithm for creating hash functions and is currently not recommended for use in secure communication [4], although this recommendation is not heeded by many as MD5 is still available for use on websites to verify the integrity of downloads and content. SHA has many different forms, starting with SHA (SHA-0) and progressing through SHA-1, SHA-224, SHA-256, and SHA-384 to SHA-512. These numbers for the most part represent the size of the hash value and thus there is a SHA for sizes 160, 224, 256, 384, and 512 bits. Whereas MD5 is only 128 bits, SHA was able to adapt and evolve to accommodate larger hash values and remain relevant to secure communication. This is why the SHA family is the only hash functions recommended by the National Institute of Science and Technology (NIST) [12]. The exact methodology of how the hash function creates the residual value is beyond the scope of this thesis. Table 4 shows the different SHA values for an empty text document, string = ““ or empty, and the SHA values for a text document that only contains a single blank character, string = 0x08 (in hexadecimal).

Table 4. SHA Values

Function	Value (String = "")	Value (String = " ")
SHA (SHA-0) (20 bytes)	f96cea198ad1dd5617ac08 4a3d92c6107708c0ef	92847a9f8c36a92affc65e9b fb4eb4f851b50dad
SHA-1 (20 bytes)	da39a3ee5e6b4b0d3255bf ef95601890afd80709	e5fa44f2b31c1fb553b6021e 7360d07d5d91ff5e
SHA-224 (28 bytes)	d14a028c2a3a2bc947610 2bb288234c415a2b01f82 8ea62ac5b3e42f	b265f33f6fe99bd366dae49c 45d2c3d288fdd852024103 e85c07002d
SHA-256 (32 bytes)	e3b0c44298fc1c149afbf4c 8996fb92427ae41e4649b 934ca495991b7852b855	4355a46b19d348dc2f57c04 6f8ef63d4538ebb936000f3 c9ee954a27460dd865
SHA-384 (48 bytes)	38b060a751ac96384cd93 27eb1b1e36a21fdb71114b e07434c0cc7bf63f6e1da27 4edebfe76f65fbd51ad2f14 898b95b	d654902b550e334bb6898d 5c4ab8ebe1aedc6c85368ea fe28e0f89b62a74a23e1ed2 0abbc10c02ce321266384d 444717
SHA-512 (64 bytes)	cf83e1357eefb8bdf154285 0d66d8007d620e4050b57 15dc83f4a921d36ce9ce47 d0d13c5d85f2b0ff8318d2 877eec2f63b931bd47417a 81a538327af927da3e	3abb6677af34ac57c0ca582 8fd94f9d886c26ce59a8ce6 0ecf6778079423dccff1d6f1 9cb655805d56098e6d38a1 a710dee59523eed7511e5a 9e4b8ccb3a4686

These two columns illustrate the vast difference in hash values just from a single character being added to a document and then re-hashed.

C. AVAILABILITY

Availability is the simplest of the pillars to describe. Either something is available or it is not. This could be a physical restriction in that something was physically removed or denied to a user, or indirectly, such as someone denying power to a building where the users were trying to access the computers. While there are many different ways for something to become unavailable, especially in the digital realm, there are too many to discuss within the scope of this thesis. For the purposes of this thesis, availability will refer to whether something is accessible or not. If a resource is accessible than it is available and if it is not accessible than it is not available.

D. AUTHENTICATION AND NON-REPUDIATION

Given the triangle for digital security consists of confidentiality, integrity, and authentication, when two or more of these are combined a new pillar is formed. When confidentiality and integrity are combined new pillars of authentication and non-repudiation are created. The use of a hash function value appended to the message creates a message authentication code; when RSA encryption is used in conjunction with a hash function digital signatures are created.

1. Message Authentication Code

A message authentication code (MAC) uses a keyed cipher block chaining encryption method to generate a one of a kind authentication code that is attached to the original message and sent with the original message to the recipient. The recipient then goes through the same steps with the received message to generate his own MAC and compares it to the MAC that was sent with the original message. If the MAC generated is equal to the MAC sent then the message was not changed in route to the recipient and the recipient knows that only someone with the MAC-key could have generated the MAC and sent the message.

2. Digital Signatures

Similar to a message authentication code, a digital signature adds a refinement to the list of people that could have sent a message or data. With a MAC, anyone that has access to the MAC-key could have generated the message but with a digital signature there is only one person that could have created the signature and this results in non-repudiation being added to the authentication of a message or data; that is, the ability to prevent the originator of a message to later retract or repudiate having sent the message.

The generation of a digital signature is similar to the generation of a MAC. The message is hashed, using one of the approved hash functions previously mentioned, and then the hash value is encrypted using the sender's private key, which was formulated using the RSA encryption previously discussed. This encrypted hash value is attached to the message and then the message is encrypted with the recipient's public key and sent to the recipient. The recipient uses her private key to decrypt the message, thus preventing anyone else from decrypting it and modifying the message en route, and the public key of the sender to decrypt the hash value sent within the signature, thus ensuring only the originator could have generated the hash. The recipient hashes the original message sent and compares that hash value to the hash value that was decrypted. If the two match then the message is authentic and the person whose public key was used to encrypt the signature could only have sent it, so there is non-repudiation. Figure 8 shows an example of how this would work.

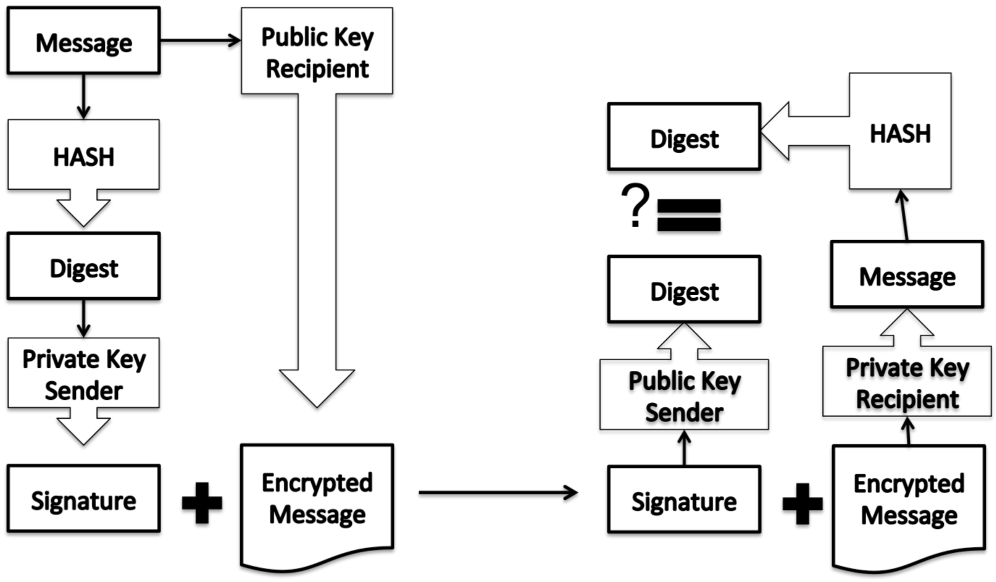


Figure 8. Digital Signature Example

This provides confidentiality, integrity, authentication and non-repudiation for the message.

Similar to digital signatures there is also Public-Key Infrastructure (PKI), which is more of an extension of or implementation of digital signatures on a large scale. PKI handles the key exchange for the digital signatures as well as has the ability to have a trusted agent verify that a signature is authentic. The way that the signature is authenticated is by using a trusted Certificate Authority (CA) to host the public keys of supported entities. For example, a user, Rebecca, generates a key pair and stores the private key but sends the public key to the CA. The CA then issues a certificate stating that the public key belongs to Rebecca and this is included when Rebecca sends her public key to Heather so that Heather has a reassurance that the public key that she received is actually Rebecca's. PKI uses X.509 as its standard for identification purposes and authentication of user submitted keys [13].

X.509

X.509 is the standard for generating a certificate from the CA to be used in the PKI system [13]. This standard tells about the user that is providing the certificate as well as how the CA generated the certificate. There are three fields to X.509 entities: the Certificate, Certificate Signature Algorithm and the Certificate Signature. The Certificate is comprised of 10 subfields. Those subfields are as follows for a X.509 version 3 certificate.

(1) Version

This describes what version of X.509 the certificate meets. There are three versions of the certificate, versions 1, 2, and 3; with 3 being the most current [14].

(2) Serial Number

This is a unique positive integer assigned by the CA to each individual certificate and no two certificates should have the same serial number [14].

(3) Algorithm ID

The algorithm used to sign the certificate is named in this section and this must match the algorithm named in the Certificate Signature Algorithm [14].

(4) Issuer

The Issuer is the name of the CA or person that is providing the certificate. The certificate can be self-signed, meaning that the user who created the public key is also the one who created the certificate and no outside entity verified the public key. This is covered in Chapter III [14].

(5) Validity

The CA puts the expiration date of the certificate in this field. This helps with revocation of any certificates and ensures that users have the most up-to-date certificates for security purposes [14].

(6) Subject

The user associated with the public key is named in this section [14].

(7) Subject Public Key Information

The type of key (RSA, DSA, or D-H) as well as the actual public key is located in this section [14].

(8) Unique Identifiers

This is used in the case that the Subject or issuer names have to be reused over time [14].

(9) Extensions

This section allows for a certification hierarchy [14]. A certification hierarchy is a list of trusted Subjects that allow for unknown users to trust a certificate and the public key. Keeping with the example of Rebecca and Heather, a certification hierarchy could be that Rebecca tells Heather to trust a public key from Denise even though Heather does not know Denise. Heather trusts Rebecca and Rebecca trusts Denise; thus, a hierarchy of trust is created, also referred to as transitive trust.

The Certificate Signature Algorithm field contains the algorithm used to sign the certificate by the CA [14]. The Certificate Signature is the value of the certificate using the Certificate Signature Algorithm. The recipient uses this to verify the integrity of a received certificate.

E. SUMMARY

In this chapter, the core concepts of confidentiality, integrity, and availability, along with the derived concepts of authentication and non-repudiation were explained. Ways in which to ensure that these pillars of secure communication are afforded and enacted were also covered. Chapter III covers how MAST is lacking in these areas and how the correction of these areas is addressed by incorporating different libraries of code into the java applications that comprise MAST.

III. PROPOSED SOLUTION

As highlighted in [1], MAST is deficient in encryption and secure communication. This presents a problem for the use of MAST on the DOD computer systems and networks. This chapter examines some of the network centric issues of MAST.

A. SECURITY ISSUES OF MAST

For a more in depth discussion of the issues described herein, refer to [1]. This thesis will address the issues included in Table 5 that is extracted from [1] and included here for ease of reference; it is from the development findings in [1]. This section of [1] has 42 of the 77 identified issues in MAST. The table is divided into three columns: the severity of the issue, the portion of the STIG that is applicable, and a detail of the requirement and how MAST fails to meet the requirement. Per [1], the severity of a finding is divided into high, medium, and low. A high severity is any vulnerability the exploitation of which will directly and immediately results in loss of confidentiality, availability, or integrity. A medium severity is any vulnerability the exploitation of which has a potential to result in loss of confidentiality, availability, or integrity. A low severity is any vulnerability, the existence of which degrades measures to protect against loss of confidentiality, availability, or integrity.

Table 5. ASD STIG Findings, from [1]

STIG ID Severity	Requirement from ASD STIG	Finding Details from SCA
APP3250 High	The designer will ensure data transmitted through a commercial or wireless network is protected using an appropriate form of cryptography.	Encryption is not used by the application.
APP3150 Medium	The designer will ensure the application uses the Federal Information Processing Standard (FIPS) 140–2 validated cryptographic modules and random number generator if the application implements encryption, key exchange, digital signature, and hash functionality.	Encryption is not used by the application.
APP3170 Medium	The designer will ensure the application uses encryption to implement key exchange and authenticate endpoints prior to establishing a communication channel for key exchange.	Encryption is not used by the application.
APP3260 Medium	The designer will ensure the application uses mechanisms assuring the integrity of all transmitted information (including labels and security parameters).	Application does not use any kind of integrity mechanism.
APP3300 Medium	The designer will ensure applications requiring server authentication are PK-enabled.	Application does not have server authentication mechanism. Lack of capability is a finding. PKI waiver required to continue without PKI enabled.

STIG ID Severity	Requirement from ASD STIG	Finding Details from SCA
APP3700 Medium	The designer will ensure unsigned Category 1A mobile code is not used in the application in accordance with DOD policy.	Application's SIMware modules contain various forms of mobile code, which are unsigned.
APP3710 Medium	The designer will ensure signed Category 1A and Category 2 mobile code signature is validated before executing	Application performs no validation of mobile code.

Seven vulnerabilities, as highlighted in Table 5, are addressed in this chapter with a combination of proposed solutions to solve the issues mentioned in [1].

B. DIGITAL SIGNATURES

Using digital signatures is the first layer of a proposed defense-in-depth plan for MAST. As discussed in Chapter II, a digital signature allows for authentication and non-repudiation of data. The proposed solution for digital signatures in support of MAST's defense-in-depth plan leverages java's security package [15], specifically the portion of the security package that handles access control, such as key pair generation, private key generation, public key generation, and random number generator. This would be used for communication between the MAST SG Server, SE Server, Graphical User Interface, and the clients or ENs. The code fragments in this section are included to help understand concepts and do not directly correlate to the actual code used in MAST; they are intended for explanatory and illustrative purposes only.

1. Digital Signatures

The digital signatures generator in the java security package generates signatures in accordance with the Digital Signature Algorithm (DSA), RSA digital signature algorithm, and the Elliptical Curve Digital Signature Algorithm (ECDSA). The proposed solution uses the java methods to create a public and private key for each of the components of MAST and store the keys in a key store that is accessible by all the components.

a. Key Generation

Java accomplishes the generation of keys via its security library. The library must be imported before the functionality can be used:

```
import java.security.*;
```

This command will import every method that is contained within the security library. Once the security library is imported, access is granted to methods that are required to create the keys for each part of the MAST system. The generation of the keys is simple in coding terms but a lot of work is done by the library routines.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance          (“ECDSA”);
```

Here **keyGen** is created as an instance of **KeyPairGenerator** using the RSA standard and the key generation algorithms that are proprietary to Sun Microsystem (now Oracle, Inc.). There is a list of different providers that could be used for the **KeyPairGenerato**r in [16]. After the instance of the **KeyPairGenerator** is created, a random number must be used to seed the key. Getting an instance of the **SecureRandom** method generates the random number.

```
SecureRandom random = SecureRandom.getInstance(“SHA1PRNG,” “SUN”);
```

The **SecureRandom** instance random meets the FIPS 140–2 requirements for randomness [17], specifically called out in Section 4.9.1. The next step is to initialize the key generator with the random number that was just generated.

```
keyGen.initialize(1024, random);
```

Here the **KeyPairGenerator** instance, **keyGen,** calls the class method, **initialize,** which instantiates the keyGen with the key size and the random number for the keys**.** The KeyPair class in the Security library stores the public and private keys generated by the KeyPairGenerator. The KeyPair instance is created using the **generateKeyPair** method from the keyGen created above. The use of “SG” in the naming of the key instances is to make it explicitly clear that these keys are associated with the Scenario Generation server. Similar naming conventions will be used for the other MAST entities.

```
KeyPair pairSG = keyGen.generateKeyPair();
PrivateKey privSG = pairSG.getPrivate();
PublicKey pubSG = pairSG.getPublic()
```

The next part of digital signatures is to use the keys once they have been created. The proposed solution generates the keys for all components involved, the Scenario Generation Server (SG), the Scenario Execution Server (SE), and the client or End Node (EN). The public keys that are generated for each component are stored on the corresponding component that will need the public key. This example provides a possible generation of the SG public and private key; the process can be repeated for the SE and all ENs that are needed. The component that is providing its signature will only need to use the private key of the key pair; continuing with this example, the SG associated keys will be used. The SG will create a signature:

```
Signature signSG = Signature.getInstance("SHA512withECDSA,");
```

The Signature class of the Security Library again uses the designated algorithm and desired provider; the two must match the key type that was generated. Now the SG has an ECDSA signature algorithm to use when communicating with the SE.

b. Using Signatures

The signature instance must be modified each time data is to be certified by the signature. For this solution the messages will be sent into a byte-wise loop that takes each byte array read from the associated file and updates the signature instance accordingly. Note that the signature instance is initialized prior to the generation of the new signature. Upon complete processing of the intended message, the signature instance is finalized with the call to the signature method, **sign()**.

```
signSG.initSign(privSG);
FileInputStream SGfis = new FileInputStream(fileName);
BufferedInputStream SGbufin = new BufferedInputStream(SGfis);
```

```
byte[] SGbuffer = new byte[1024];
  int len;
  while ((len = SGbufin.read(SGbuffer)) >= 0) {
    signSG.update(buffer, 0, len);
  };
  SGbufin.close();
  byte[] realSig = signSG.sign();
```

The subject of the signature is fileName, which is the file that is going to be sent from the SG to the SE. This file is a new scenario file that is to be sent from the SG to the SE and for execution by SE. Now that the signature has been created, it will be attached to the associated data object that is to be sent between SG and SE. With the signature, SE will be able to verify that the integrity of the new scenario was not compromised during the communication with SE.

c. *Verifying Signatures*

The process for generating a signature must be completed by the recipient, using the data sent by the source (in this case the SG), to enable the recipient to validate the received data integrity. The same process will work for any component of MAST, so the communication between SG and SE is verified in the same manner as the communication between SE and EN.

The receiver must have the public key of the sender in order to verify the signature. The acquisition of that key is discussed separately in the secure socket layer / transport layer section of this chapter. The first step in verifying the data integrity is to load the bytes of the data that was signed. Here it is assumed the recipient stored the received data (a serialized Java object) in a file, thus the file access methods of Java can be leveraged.

```
    FileInputStream SEsigfis = new FileInputStream(fileName);
    byte[] sigToVerify = new byte[SGsigfis.available()];
    SEsigfis.read(sigToVerify );
```

```
   SEsigfis.close();
```

Next the recipient must create a signature instance to be used to verify the integrity of the data received and associated with a digital signature. The instance must be initialized before use.

```
Signature SEsig = Signature.getInstance(“ECDSA”);

   SEsig.initVerify(SGpubKey);
```

Now the receiver has to take the message that was sent and generate its own version of the signature using the public key from the sender. This is accomplished in the same way that the sender created the initial signature using the sender’s private key.

```
FileInputStream SEdatafis = new FileInputStream(fileName);

   BufferedInputStream bufin = new BufferedInputStream(SEdatafis);

   byte[] buffer = new byte[1024];

   int len;

   while (bufin.available() != 0) {

     len = bufin.read(buffer);

     SEsig.update(buffer, 0, len);

     };

bufin.close();
```

Finally, the last step is to compare the two signatures using the class method, **verify()**, which returns a Boolean variable indicating whether or not the data that was received is valid. If **False** is returned the recipient must discard the received data. Further action might include requesting a retransmission by the sender as well as alerting the user that there is a possible compromise of the network. The latter might be conditioned upon the receipt of multiple invalid signatures.

```
boolean verifies = SEsig.verify(sigToVerify);
```

2. DOD RMF Vulnerabilities Addressed through Digital Signatures

The use of digital signatures helps to ensure that there is integrity throughout the system and that the message that is being sent has not been tampered with while en route between the two entities that are communicating. Since the digital signatures are only between the communicating portions of MAST and do not directly verify outside sources of code or communication and it does not use encryption, there is very little of the DOD RMF that is corrected by the addition of digital signatures alone. The two parts corrected are shown in Table 6.

Table 6. DOD RMF Vulnerabilities Addressed by Digital Signatures, from [1]

APP3700 Medium	The designer will ensure unsigned Category 1A mobile code is not used in the application in accordance with DOD policy.	Application's SIMware modules contain various forms of mobile code, which are unsigned.
APP3710 Medium	The designer will ensure signed Category 1A and Category 2 mobile code signature is validated before executing	Application performs no validation of mobile code.

As Table 6 shows, the vulnerabilities of MAST with regard to validation are of medium severity. MAST should now be able to validate communication between components. Now that SG can upload and send a scenario to SE, the use of a digital signature becomes a critical requirement more so than before when this capability did not exist. The signing of the scenario by SG before sending and the verification of the scenario before using allows for MAST to address these two vulnerabilities as mentioned in [1]. The remaining problems addressed by the DOD RMF are handled by the addition of SSL to the system.

C. SECURE SOCKET LAYER / TRANSPORT LAYER SECURITY

TLS is used to secure many different types of information transfers. Examples include web browsing, financial transactions, healthcare transactions, Virtual Private Networks (VPNs), and custom protocols. It is a layered and application independent protocol, typically operating between layer 4 and 5 of the TCP/IP Stack. Its primary goal is to provide two communicating applications with privacy and data integrity [18], but it can be used in other ways, such as in VPNs, transactional systems and transfer protocols as mentioned above, where TLS is optionally offered at lower levels of the TCP/IP Stack.

The SSL protocol was originally designed by the Netscape Corporation and used in a client-server relationship. SSL 1 was never officially released. SSL 2.0, however, was officially released in February of 1995. Shortly thereafter, in 1996, SSL 3.0 was released due to known security flaws in SSL 2.0 and was subject to a complete redesign of the protocol. Though Netscape released SSL 3.0, it was historically documented in [19] because TLS 1.0 uses SSL 3.0 as a foundation.

TLS 1.0, as defined by [20], builds upon the SSL 3.0 design, and can optionally negotiate to downgrade to SSL 3.0 for legacy interoperability. TLS 1.1, as defined in [21], was formalized in April 2006. It was released primarily in response to weaknesses in the initialization vector selection and padding error processing found in SSL 3.0 and TLS 1.0 [21]. While TLS 1.1 and 1.2 were minor changes to protocol policies, TLS 1.2 added enhanced cryptographic modifications and many other improvements as specified in [18]. Of note is the transition from MD5/SHA-1 to SHA-256 in several components of the suite and the depreciation of cipher suites such as IDEA and DES.

1. SSL/TLS Technical Components

TLS is not a simple stand-alone protocol. It is instead an aggregate of multiple sub-protocols and technologies described in greater detail below. Since the goal of TLS is to provide confidentiality, integrity, and availability and since other tried and true technologies already exist to provide some proper subset of these characteristics, TLS implements an appropriate methodology of combining them. It leverages their strengths and account for their weaknesses with other sub-protocols and methodologies. The end product provides the desired security for communication.

a. TLS Handshake

The Handshake is used to negotiate security parameters for the session. Since there are many different options for a particular TLS session, both hosts need to agree upon which cipher suite—key establishment, digital signature, confidentiality and integrity algorithms, etc.—will be used [22]. This portion of TLS has attracted the most attention, as it has been the primary vector of attack, typically involving downgrading to legacy cipher suites. The TLS Handshake is a combination of three components: handshake, change cipher spec, and alert.

(1) The Handshake (Hello)

The TLS Handshake, specifically the “Hello” portion, is designed to negotiate session parameters (cipher suite). The client informs the server of the protocols and standards that it supports and then the server selects the highest common protocols and standards. Specifically, the Client Hello message specifies a supported cipher suite that is defined by a key exchange algorithm, a bulk encryption algorithm, a MAC algorithm, and a Pseudo Random Function (PRF) [18], MACs and PRFs are discussed later in this report. Once the server has selected a cipher suite, it informs the client through the Change Cipher Spec.

(2) Change Cipher Spec

The server notifies the client of the security parameters for the remainder of the session using the Change Cypher Spec protocol, as decided upon during the initial handshake. Once the client is in receipt of the server’s Change Cipher Spec message, it sends its own Change Cipher Spec message to acknowledge and confirm that it will continue with the selected parameters. The Finish messages are encrypted using the agreed upon security parameters and serve as an integrity check to the handshake.

(3) Alerts

Alerts are messages that convey errors and warnings such as unexpected message, record overflow, or bad certificate. Those that are considered “fatal” cause the immediate tear down of the session to prevent a compromise in security.

b. Putting it All Together

The handshake is one of the most important, and as previously noted vulnerable, components of the overall TLS protocol. During the handshake, the two hosts (typically a server and a client) negotiate algorithms, cipher suites, symmetric keys, and other session parameters.

Though the precise steps the handshake protocol executes will depend on the server and client configurations and compatibility, it can be generally viewed to follow this procedure; a graphical representation of the TLS Handshake is depicted in Figure 9 [23].

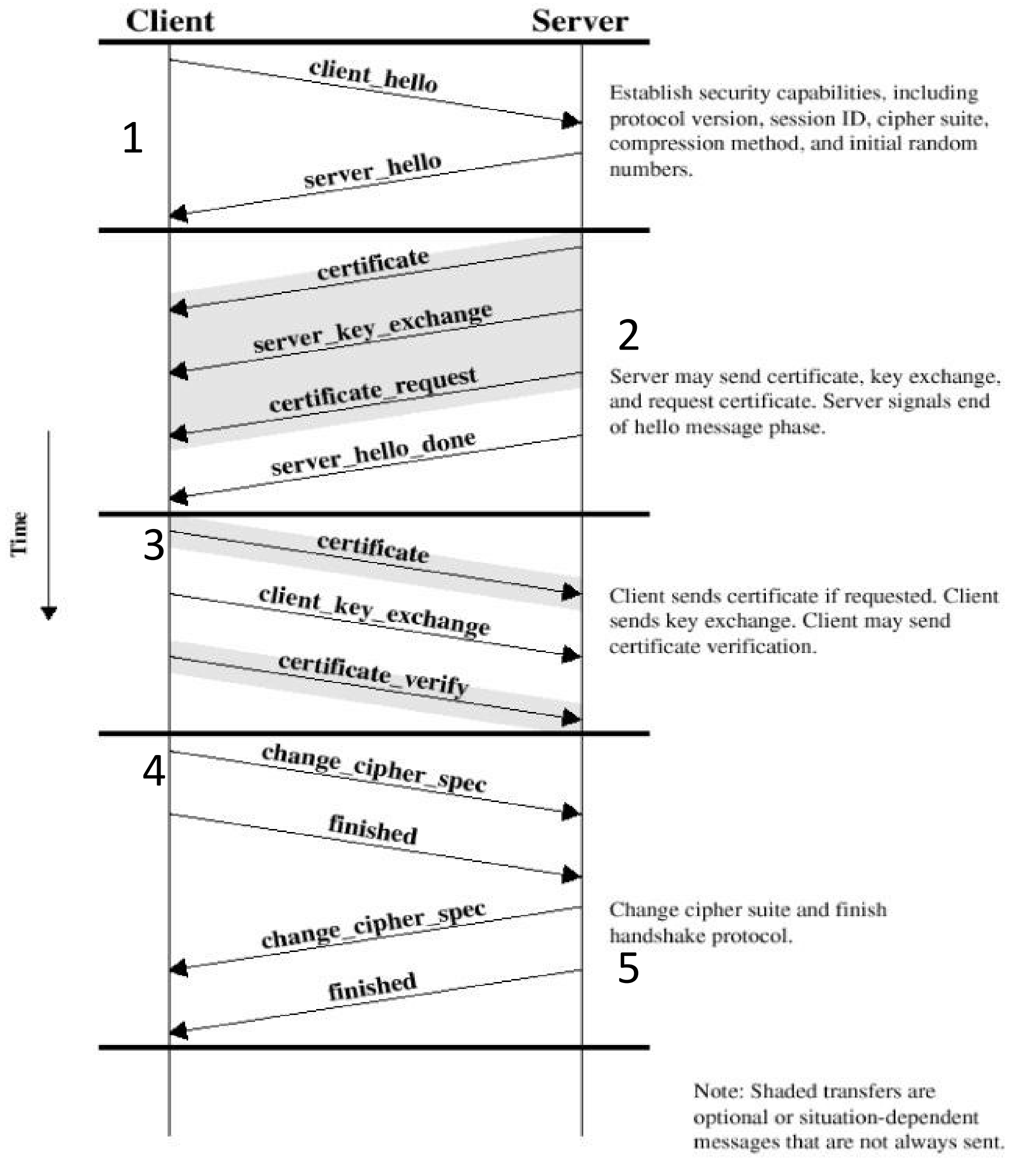

Client
Server
client_hello
server_hello
1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.
certificate
server_key_exchange
certificate_request
server_hello_done
2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.
Time
3
certificate
client_key_exchange
certificate_verify
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.
4
change_cipher_spec
finished
change_cipher_spec
finished
5
Change cipher suite and finish handshake protocol.
Note: Shaded transfers are optional or situation-dependent messages that are not always sent.

Figure 9. Graphical Representation of a Typical TLS Handshake, from [23]

The corresponding numbered list helps expand on what exactly is being explained in Figure 9 the numbers on the Client and Server sides of the figure relate to the numbers on the list.

1. “Client Hello: The client sends a ClientHello message to which the server must respond with a ServerHello message, or else a fatal error will occur and the connection will fail. The ClientHello and ServerHello are used to establish security enhancement capabilities between client and server. This message exchange establishes the following attributes: Protocol Version, Session ID, Cipher Suite, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random.”[18]
2. Server HelloDone: “Following the hello messages, the server will send its certificate in a Certificate message and the Server Key Exchange message ... If the server is authenticated, it may request a certificate from the client, if that is appropriate to the cipher suite selected. Next, the server will send the ServerHelloDone message, indicating that the hello-message phase of the handshake is complete. The server will then wait for a client response.” [18]
3. Optional Client Certificate: “If the server has sent a CertificateRequest message, the client MUST send the Certificate message. The ClientKeyExchange message is now sent, and the content of that message will depend on the public key algorithm selected between the ClientHello and the ServerHello. If the client has sent a certificate with signing ability, a digitally-signed CertificateVerify message is sent to explicitly verify possession of the private key in the certificate.” [18]
4. Client Change Cipher Spec + Finish: “At this point, a ChangeCipherSpec message is sent by the client, and the client copies the pending Cipher Spec into the current Cipher Spec. The client then immediately sends the finished message under the new algorithms, keys, and secrets.” [18]
5. Server Change Cipher Spec + Finish: “In response, the server will send its own ChangeCipherSpec message, transfer the pending to the current Cipher Spec, and send its Finished message under the new Cipher Spec.” [18]

c. Shared Secret Negotiation

Though technologies and techniques have changed drastically over time, it has typically not been difficult to achieve confidentiality as discussed below, especially with the aid of modern computers. The greatest challenge has been to devise a method by which one can transfer the encrypted material in such a way that it is easy for the intended recipient to read but difficult or impossible for an adversary to do so. For example, if using AES, one can encrypt a message such that it is computationally infeasible to decrypt the message without the key. Then again since AES is a symmetric encryption algorithm, the sender must somehow convey the key to the intended recipient, and do so securely. However, herein lays the problem: if the sender already had a secure channel to transmit the key, then he would not be seeking another channel. The solution to this problem is PKI.

PKI uses the concept of asymmetric cryptography wherein a message is encrypted with one key and decrypted with another. While the details of generic PKI were discussed in Chapter II, the assumption going forward is that PKI provides a mechanism by which a sender can convey a secret to another party where both sender and receiver are guaranteed one another's identities and eavesdropping by a malicious party is impossible. However, this method of communication is extremely computationally expensive and becomes infeasible for substantial communication, particularly on embedded or mobile devices.

The specific PKI protocols utilized by TLS are RSA, DH, and Elliptic Curve DH (ECDH). Specifically, the key exchange algorithm may be selected from the following list for TLS 1.2: RSA, RSA_PSK, DHE_RSA, ECDHE_RSA, DHE_DSS, DH_DSS, DH_RSA, ECDH_ECDSA, ECDH_RSA, ECDHE_ECDSA [18].

(1) Confidentiality

Using a symmetric encryption algorithm provides confidentiality. As mentioned above, symmetric encryption alone presents many challenges. However, after Secured Secret Negotiation to authenticate and open a secure channel between hosts, a symmetric key can safely be shared. After exchanging a symmetric key (client and server encryption keys derived from the "master secret"), the session can benefit from the performance benefit of symmetric encryption.

TLS 1.2 defined in [18] selects a "Bulk Cipher Algorithm" from RC4, 3DES, or AES. However, [24] prohibits the use of RC4 in all TLS versions, [25] adds the ARIA cipher suites, and [26] adds the Camellia cipher suites.

(2) Integrity

While Secure Secret Negotiation and confidentiality are critical, and defend against a wide array of malicious threats, they are insufficient to guarantee CIA without data integrity. To provide integrity, TLS uses two MAC keys, one for when the server is sending (server write key) and one for when the client is sending (client write key). The MAC keys are derived from the shared master secret. Both server and client are in possession of these keys. When a sender is transmitting a message, it first calculates the MAC of the message with the appropriate key and then encrypts both the message and the MAC. The resultant cipher-text is transmitted to the receiver.

The receiver decrypts the cipher-text using the appropriate symmetric key. The receiver then uses the appropriate MAC key to generate a MAC of the message – if it matches the attached MAC, then the message has not been tampered with or corrupted in transit.

The MAC function is either the one specified by the chosen cipher suite and applied as described above, or it is implicitly provided by Authenticated Encryption with Associated Data (AEAD) cipher mode in which case the client and server write keys are unnecessary. The most common implementations of TLS use a MAC from the chosen cipher suite and not utilize the AEAD functionality; therefore it will not be expanded upon here. The available MAC algorithms for TLS 1.2 are: hmac_md5, hmac_sha1, hmac_sha256, hmac_sha384, and hmac_sha512 [18].

(3) Authentication

Though already discussed above, authentication is a critical component of the TLS protocol. Without some method of effective authentication, the protocol would be subject to man-in-the-middle and other related attacks. The Shared Secret Negotiation already provided a mechanism for authentication. Due to the nature of PKI, the client was able to deem the server authentic when its messages arrived and were decipherable. That is because the message was sent after being encrypted with the server's private key. When the client decrypted the message with the server's public key, he knew that the server was the only possible sender of the message. The same process works in reverse should the server require that the client authenticate using its own certificates.

(4) Anti-Replay

A sequence number provides for anti-replay. The sequence number is bundled with the message, prior to the application of the MAC. It is a monotonically increasing number that, if seen out of order or duplicated, can detect the presence of replay attacks.

(5) Key Management

Key management covers several topics, most outside the scope of this paper. The aspect of key management most relevant to this thesis is that of the PKI infrastructure. While the usage of PKI has been discussed, the distribution of keys has not. The most prevalent and robust method by which PKI is employed for TLS is through certificate authorities (CAs). When a user goes to a website, for example, which utilized TLS, such as a bank, he or she needs assurance that the certificate being presented in fact belongs to that institution and not that of an undesired third party.

It would be unreasonable to expect each institution to manually distribute its public keys to all of its customers. Similar challenges exist in most other applications of TLS. The current paradigm is to maintain several “root” CAs, such as Verisign or Digicert, who conduct varying levels of background investigations to validate the identify of an organization or individual. Once they do so, they certify the organization’s credentials by issuing them a certificate signed by the root CA’s certificate. Since there is a short list of root CAs, the major browser and application makers include the root CA’s public keys within the application. When a user is presented with a website’s certificate, it needs only check that the signature on its certificate validates against the root CA’s public certificate. Thus, the user is assured that the certificate being presented is authentic. In most applications, this process is automated until a violation of the protocol is encountered. This automation of authenticity is the portion of TLS that provides the Availability of CIA and is largely responsible for the sweeping success and ubiquity of TLS.

2. Creating a SSL Socket

As with digital signatures, java provides code libraries to support secure sockets. These libraries are imported by applications through the following code:

```
import javax.net.ssl.SSLServerSocket;

import javax.net.ssl.SSLServerSocketFactory;

import javax.net.ssl.SSLSocket;
```

These imports allow for the creation of a SSL Server Socket Factory, SSL Server Socket, and a SSL Socket. The SSL Server Socket Factory is used to initialize the keys and certificates for the SSL Server Sockets that are created. The SSL Server Socket Factory, when initialized, has the full array of possible algorithms available for securing the connection. The initialization is achieved through the command:

```
SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();
```

The .getDefault() method for the SSLServerSocketFactory has to access a key store that includes the private keys and the trusted members to associate with any SSL Server Sockets that are initialized. The private keys are for the SSL Server Sockets and the list of trusted members are the public certificates of CAs. The inclusion of the CAs allows for the peer authentication trust decisions. Since MAST should only allow for communication between its components, the default key store is over written and only the private keys and trusted members of MAST are loaded. MAST does this with a system call to replace the default key store.

```
System.setProperty(“javax.net.ssl.keyStore,” “MAST_Keystore”);

System.setProperty(“javax.net.ssl.keyStorePassword,” “abc123”);
```

Using sslServerSocketFactory a new SSL Server Socket is created and initialized by passing the port on which the SSL Server Socket is to listen and the maximum number of clients that the SSL Server Socket should accept.

```
sslServerSocket = (SSLServerSocket)

                    sslServerSocketFactory.createServerSocket(listen_port, max_clients);
```

As [27] and [28] explain, there are vulnerabilities in some of the algorithms inherent to SSL. MAST hardens the SSL Server Socket before it is used.

```
hardenSSLServerSocket(sslServerSocket);
```

Hardening of the SSL Server Socket via hardenSSLServerSocket allows for MAST to remove the vulnerable algorithms from the SSL Server Socket and ensure that whenever a connection is made to the SSL Server Socket that no vulnerable algorithms are used. The call hardens the specified SSL Server Socket by disabling weak protocols and cipher suites. This is done by requesting supported suites and protocols and then disabling those that are prohibited as specified in the static variables for the class. The now hardened SSL Server Socket is set to accept incoming connections on the listening port provided during creation.

A client SSL Socket is created similarly to a SSL Server Socket. The calls to import the necessary libraries are:

```
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
```

The SSLSocketFactory is initialized using a similar method call as that used for the SSLServerSocketFactory and then an SSLSocket can be created from the SSLSocketFactory. After creation, the SSLSocket is bound to a port by the operating system; the exact port to which the socket is bound is not important since the client is not listening for incoming traffic.

3. Using a SSL Socket

The method of using a SSL Socket is the same as a normal non-secure socket. First, a server socket on the server and a socket on the client are created, as described above. Since the server socket is bound to a port the client socket must use that port number to contact the server. For the initial connection the server and client will go through the hello and handshake to establish a secure connection, as described above.

4. DOD RMF Vulnerabilities Addressed through SSL

The creation of a SSL connection encrypts the communication between the components of MAST. The SSL session ensures that the two entities communicating are in fact the entities that are supposed to be communicating. With the addition of encryption and authentication between server and client, five of the seven CIA-associated vulnerabilities identified by [1] as pertaining to MAST are addressed. As Table 7 shows these vulnerabilities are addressed via the use of the SSL connection. The connection creates an encrypted pipeline between components. The only way the pipeline is created is through the use of PKI that in turn ensures proper authentication of the end points of the pipeline.

Table 7. DOD RMF Vulnerabilities Addressed by SSL / TLS, from [1]

APP3250 High	The designer will ensure data transmitted through a commercial or wireless network is protected using an appropriate form of cryptography.	Encryption is not used by the application.
APP3150 Medium	The designer will ensure the application uses the Federal Information Processing Standard (FIPS) 140–2 validated cryptographic modules and random number generator if the application implements encryption, key exchange, digital signature, and hash functionality.	Encryption is not used by the application.
APP3170 Medium	The designer will ensure the application uses encryption to implement key exchange and authenticate endpoints prior to establishing a communication channel for key exchange.	Encryption is not used by the application.
APP3260 Medium	The designer will ensure the application uses mechanisms assuring the integrity of all transmitted information (including labels and security parameters).	Application does not use any kind of integrity mechanism.
APP3300 Medium	The designer will ensure applications requiring server authentication are PK-enabled.	Application does not have server authentication mechanism. Lack of capability is a finding. A PKI waiver would be required to continue without being PKI enabled.

D. SUMMARY

Chapter III addressed seven vulnerabilities of the total 77 identified by [1]. The addressed vulnerabilities all focused on confidentiality, integrity, authentication, and non-repudiation. The proposed solutions also address these issues. The use of a SSL connection ensures that a secure pipeline is established between components of MAST and only intended participants are able to communicate. The addition of digital signatures ensures that whatever is received through the pipeline is exactly what is supposed to be in the pipeline. Further work is needed to address the remaining 70 open findings from [1]. Thirty-five identified issues still remain within the development portion of the findings in [1]. Twenty-five out of the 70 outstanding issues are with regard to the deployment of MAST, 12 of the 70 pertain to program management, and 10 of the 70 are dealing with testing; these remaining issues do not rely on communication between any components of MAST and have to do with project management requirements. A major hindrance to the proposed solution is the current implementation of MAST. While a functional work around is used, a better solution would be achieved with a restructuring of the current MAST implementation. Chapter IV addresses the testing of the proposed solutions.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. TESTING

A. ENVIRONMENT PREPARATION

Below is a description of the environment in which MAST was evaluated and exploited to demonstrate inherent vulnerabilities in the current MAST implementation. First, a listing of relevant software is provided, followed by the configuration necessary to run the virtual environment and conduct the exploit.

1. The Virtual Environment

The hypervisor used was VirtualBox version 4.3.26_Ubuntu r98988. Though other hypervisors were available, VirtualBox offered the simplest installation and configuration, along with a straightforward virtual network configuration.

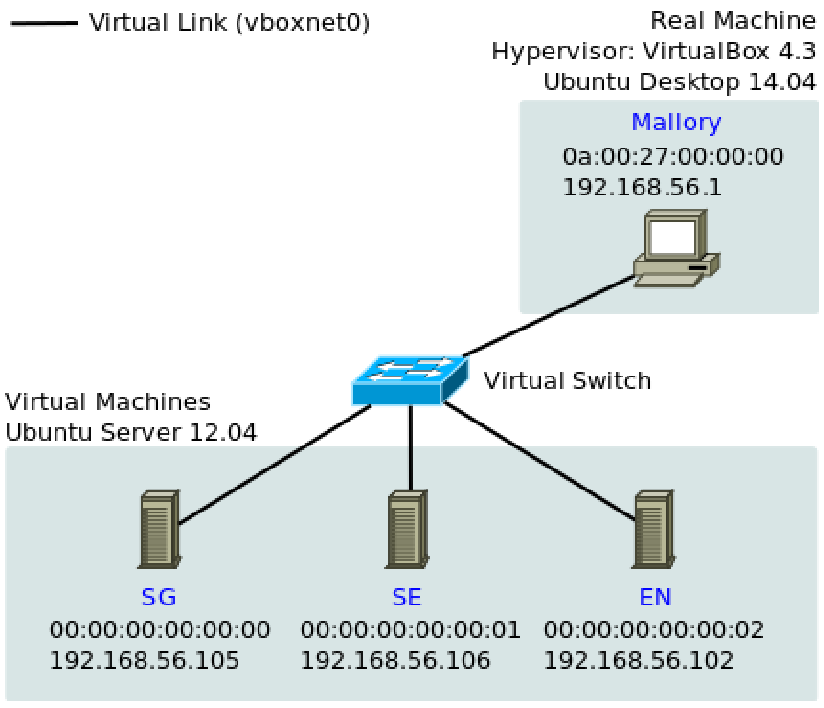


Figure 10. Virtual Environment

The virtual host/hypervisor ran Ubuntu Linux 15.04 as seen in Figure 10. Ubuntu Linux was chosen due to the author's familiarity with the operating system and its ease of setup for this kind of testing. Ubuntu 15.04, specifically, was chosen to leverage the latest VirtualBox and Linux Kernel versions to support the hypervisor role.

The virtual guests ran Ubuntu 14.04.2 as shown in Figure 10. Ubuntu 14.04.2 was selected as it is the latest 'stable' release of Ubuntu Linux. Although the preponderance of DOD end-systems are Microsoft Windows hosts, the relevant MAST components are not only written in Java but are also completely operating system independent. Thus, an exploit in this environment *should* be similarly effective when used with any operating system. It is beyond the scope of this thesis to verify this similarity; however, replication of the test execution using Microsoft operating systems as the guest hosts would allow for validating that expectation.

2. Configuration

The following sections describe the necessary configuration of the different pieces of software used in this project. The descriptions are written assuming a working knowledge of basic Linux administration and an understanding of how virtual machines work. In all of the following descriptions, a **sudo apt-get install** on a Debian or Ubuntu system, combined with the listed package, should be sufficient to install the required software *and* their dependencies. All other configuration instructions should also work on a Debian-based system given the right packages are installed.

a. The Host/Hypervisor

The hypervisor ran Ubuntu Desktop 15.04 with Linux Kernel version 3.19.0-21-generic (SMP) x86_64. The packages required for the hypervisor were:

```
ettercap-graphical virtualbox virtualbox-guest-additions-iso wireshark
```

Configuring Wireshark properly was nontrivial. The following commands applied the appropriate permissions to allow a non-root user to run Wireshark without needing root privileges (note that log-out is necessary for the group permission changes to take effect):

```
$ sudo addgroup wireshark
$ sudo adduser $USER wireshark
$ sudo chgrp wireshark /usr/bin/dumpcap
$ sudo chmod 755 /usr/bin/dumpcap
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/bin/dumpcap
```

In order to view the traffic necessary for performing this project, a shared virtual network was created. The best mode for this network was "Host-Only Network," configurable under File → Preferences → Network → Host-Only Networks. This arrangement allowed the host machine to both run the virtual machines and share their network, simplifying the execution of the project. The end state connected the host/hypervisor and all VM guests to a virtual switch.

Once the host-only network (referred to hereafter as **vboxnet0**) was configured, the VM guests were created. This was best accomplished by creating one guest virtual machine and then cloning it as described in the next section. Most defaults for creating a virtual machine were sufficient. Specifically, the first guest was created with one CPU, 192MB of RAM, and 8GB of storage space. As alluded to above, the network interface was configured to use **vboxnet0**. Though it was not strictly necessary, configuring the MAC addresses with easily identifiable values like **00:00:00:00:00:01** simplified traffic analysis.

b. The Guests

The virtual guests ran Ubuntu Server 14.04.2 with Linux Kernel version 3.13.0-55-generic x86_64. During installation, the "Minimal VM Guest" option was selected to optimize the guests as virtual machines. The packages required for the guests were:

```
openjdk-7-jre-headless build-essential
```

While the above were all of the required packages, the following packages were also installed for convenience:

```
tcpdump bash-completion vim acpid openssh-server
```

Furthermore, the VirtualBox guest-additions packages were installed on both host and guests to facilitate the exchange of files between the host and guest(s), but this step was not strictly necessary.

```
# Mount the guest iso through host GUI, create a shared folder through the
# GUI, naming it "share" and then issue the following commands:

$ sudo mount /dev/cdrom /media/cdrom
$ cd /media/cdrom/
$ sudo ./VboxLinuxAdditions.run
$ sudo mount -t vboxsf -o uid=$UID,gid=$(id -g) share /mnt
$ sudo usermod -a -G vboxsf $USER
$ ln -s /media/sf_share ~/share
```

Once the first host was configured, it was cloned rather than having to create and configure each new VM individually. Once cloned, the following final steps were taken:

- Changed the hostname appropriately in both **/etc/hosts/** and **/etc/hostname**
- Reset eth numbering by editing the file **/etc/udev/rules.d/70-persistent-net.rules** and deleting the line with the old mac address and then changing **eth1** to **eth0**.

While the above shared folder steps were not required, the following sections assume that it was done with respect to directory names. The alternative was to manually copy the required files (listed below) to a directory called **/home/$USER/share/[SG|SE|EN]**, with the appropriate module specified at the end.

This testing was completed using MAST's SVN repository version 347. The files for each MAST component were as follows: SG_347.jar, SE_347.jar, Scenario.txt (with the file listed in the paragraphs below), startServer.bat, EN_347.jar, module_list.txt (with the file listed in the paragraphs below), and startClient.bat

The scenario file that the SE had to have in order to complete this testing was named **drivebydownload.txt** and had the following contents (copied from **ScanSingleHost.txt**):

```
[Scenario]
  Name=Scan Single Host
  MinClients=1

[ModuleList]
  1=NMAP

[GroupList]
  1=1
  2=100%
  3=0

[CommandList]
  1=NMAP --unprivileged 127.0.0.1

[Events]
  1=T 4000 SGC 1 1
```

The reason for the discrepancy between the file name and file contents was that MAST was "hard-coded" to execute the **drivebydownload.txt** for testing because the functionality did not exist to run a scenario from the CLI.

The EN had the following contents in the file **module_list.txt**:

```
[module 1]
name = PING
exec = ping
[module 2]
name = NETSTAT
exec = netstat
[module 3]
name = HPING
exec = /usr/bin/sudo /usr/bin/hping3
[module 4]
name = returncode
exec = modules/testExit.sh
[module 5]
name = pwd
exec = pwd
[module 6]
name = eicar
exec = res/modules/eicar.bat
[module 7]
name = PORTSCAN
exec = res/modules/eicar.bat
[module 8]
name = JavaTest
exec = java -jar res/modules/javatest.jar
[module 8]
name = DriveByDownload
exec = java -jar res/modules/DriveByDownload.jar
[module 9]
name = NMAP
exec = /usr/bin/nmap
[module 10]
name = WinPcap
exec = res/modules/installWinPCap.bat
[module 11]
name = EmailEicar
exec = java -jar res/modules/EmailEicarModule.jar
[module 12]
name = EmailBatch
exec = java -jar res/modules/EmailBatchModule.jar
[module 13]
name = WriteEicar
exec = res/modules/WriteEicar.bat
```

The emboldened line had been modified to enumerate the directory where the nmap binary is located on a Debian system, Debian being the distribution from which Ubuntu is derived. This was the only line that needed to be changed, as NMAP was the program used by the scenario to be exploited in this project. Again, this modification would not be necessary if using MS Windows machines.

3. Running the VM Guests

Running MAST on the guests was a relatively simple process. Though order was not critical, for several reasons it was best to run the services beginning with the SG, followed by the SE, and finally the EN. Below are the specific commands to launch the MAST service on each system. Please note that changing directory to the respective jar file was necessary since the MAST program used files in the present working directory, where the configuration files detailed above were expected to be found. Note that the IP of the SG was 192.168.56.105 and the IP of the SE was 192.168.56.106 and finally the IP of the EN was 192.168.56.102.

a. Running the SG

The SG is the top node of the network tree and therefore does not need an IP address of a server to start the program.

```
cd /home/$USER/share/SG/
java -jar SG_347.jar
```

b. Running the SE

Since the SE needs to communicate with the SG for updated scenarios and also to allow for the SG to start or stop scenarios the IP address of the SG must be provided to the SE at start time, along with the port number by which to establish the connection.

```
cd /home/$USER/share/SE/
java -jar SE_347.jar 192.168.56.105 30001
```

c. Running an EN

The EN is provided the IP address and port of the SE from which it will receive commands during the execution of a MAST scenario.

```
cd /home/$USER/share/EN/
java -jar EN_347.jar 192.168.56.106 30000
```

B. EXPLOITATION

The attempt of the exploit is to stress the host system and consume resources on the host network. Ultimately the exploit is showing a proof of concept that MAST is vulnerable to attacks. The attack took place between the SE and one EN. A graphical representation of the network is depicted in Figure 11.

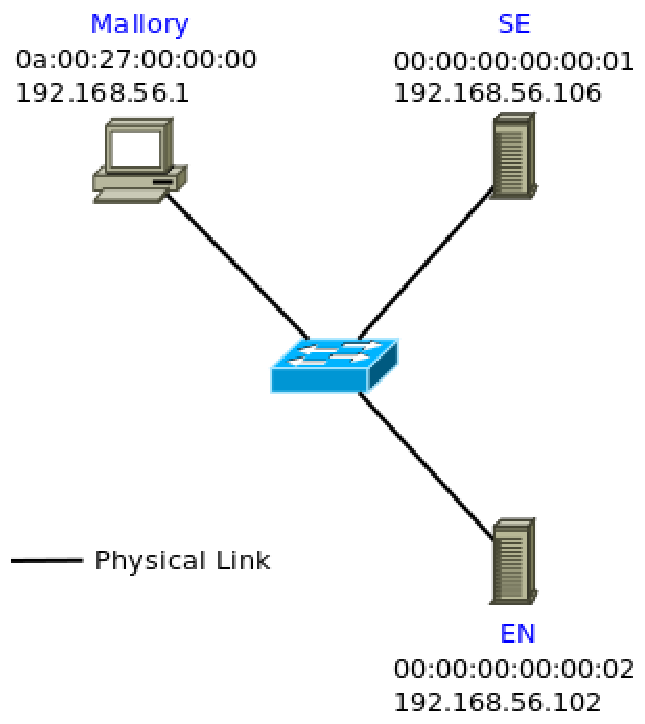


Figure 11. MAST Sample Network

1. “The Angle”

While there are many theoretical exploit ‘vectors,’ only one is needed to demonstrate the vulnerability in MAST. It is also important to note that different attack vectors have varying levels of practicality. For example, data sent in a TCP stream between two Java instances is theoretically trivial to modify ‘in flight’; however, the method by which Java bundles the data in the TCP stream makes the arbitrary modification of the data impractical. Specifically, the length of a string must be the same and cannot contain spaces or certain special characters or Java will drop the packet. Further in the chapter a packet is modified “in flight” to change the content while still maintaining the proper length.

While it is possible to programmatically modify Java TCP streams, the demonstrated attack instead leveraged the versatility, and therefore rich selection of command line arguments, of NMAP to edit strings in flight without changing their length. In this way, it was possible to demonstrate vulnerability without implementing a needlessly reusable and versatile exploit. Analogous to the mathematics arena, all that was necessary to disprove a statement such as

“Communication between MAST modules is [always] secure”

was to simply demonstrate a single counter example in the form of

“There exists a case in which communication between MAST modules is not secure.”

2. Reconnaissance and “Preparing the Battle-Space”

Some sort of traffic sniffing was necessary to view the traffic between the MAST modules. Though **tcpdump** could be used on either side, this would have been an unrealistic circumstance in practice. Thus, a tool and method was needed to perform some sort of Man-in-the-Middle (MitM) attack.

There were many tools and options from which to choose to perform a MitM attack, but **ettercap** and ARP Poisoning were chosen for simplicity and flexibility. As with any MitM attack, the goal was to redirect all traffic between two or more hosts through the attacker’s machine first, giving the attacker the ability to view and potentially modify that traffic. The ARP Poisoning attack accomplished this by poisoning the ARP tables on the victim hosts. Specifically, it aimed to replace the MAC address associated with the IP address of each victim with the MAC address of the attacker. Thus, all traffic bound for the victim machines, at the link layer, went to the attacker instead. The attacker then could drop, forward, or modify and forward all traffic between the victim hosts. Figure 12 depicts the correct and intended flow of information prior to the ARP Poisoning MitM attack.

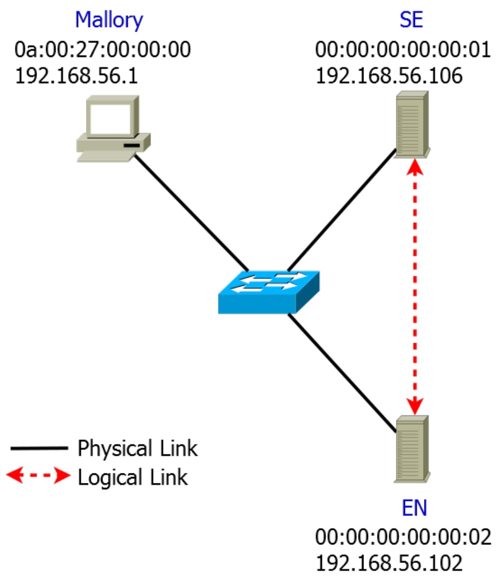


Figure 12. Correct Network Communication

To execute the attack, Mallory (the hypervisor) sent a "gratuitous" (unsolicited) ARP reply to the SE with the IP address of **192.168.56.102** and the MAC address of **0a:00:27:00:00:00**. Even though this reply was unsolicited, it was accepted nonetheless due to the insecure design of the protocol. Though this attack would not have worked in an IPv6 environment, there were other MitM attacks available for those circumstances. Next, Mallory sent an ARP reply to the EN with the IP address of **192.168.56.106** and the MAC address of **0a:00:27:00:00:00**. The result was that any traffic between the EN and the SE would be sent through Mallory. The resultant network is shown in Figure 13.

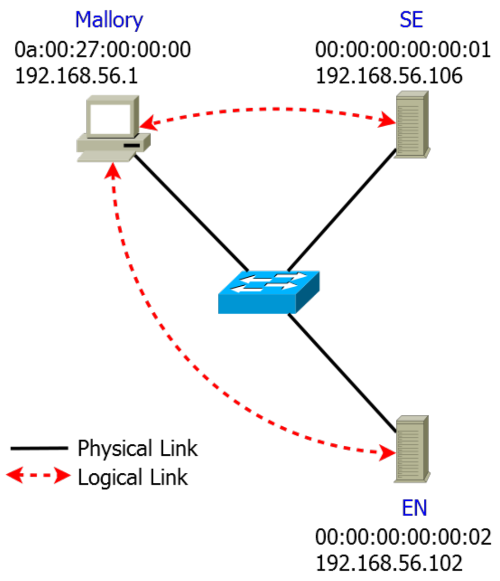


Figure 13. Compromised Network Communication

The command to execute the ARP MitM was:

```
sudo ettercap -T -q -M ARP:remote /192.168.56.106// /192.168.56.102//
```

Once traffic was flowing through Mallory, analysis of the traffic itself was possible by using Wireshark on the **vboxnet0** network. At this point, it was clear that the data between the modules was transmitted in plain text. After sifting through several conversations between the SE and the EN, and realizing that the most vulnerable module to attack was one using NMAP, it was decided to focus on the NMAP conversations.

Using the Wireshark filter **ip.src == 192.168.56.106 and ip.dst == 192.168.56.102 and tcp contains "NMAP,"** all irrelevant conversation was ignored. First, the MAST services were started, and then once the normal exchange of traffic had ceased, a capture was commenced. In the window of the SE, the command **start** was issued. As discussed earlier, the default module to run was NMAP. The expectation was that some packet would initiate at the SE and travel to the EN with the NMAP module indicated and appropriate command line options. The resultant packet can be seen in Figure 14.

```
0000  0a 00 27 00 00 00 00 00  00 00 00 02 08 00 45 00   ..'..... ......E.
0010  00 f2 de de 40 00 40 06  69 06 c0 a8 38 6a c0 a8   ....@.@. i...8j..
0020  38 66 75 30 c1 72 34 b6  e8 dc 90 d2 ce cd 80 18   8fu0.r4. ........
0030  03 e9 d8 c9 00 00 01 01  08 0a 00 01 36 98 00 01   ........ ....6...
0040  23 b9 78 70 00 00 00 02  74 00 04 4e 4d 41 50 73   #.xp.... t..NMAPs
0050  72 00 13 6a 61 76 61 2e  75 74 69 6c 2e 41 72 72   r..java. util.Arr
0060  61 79 4c 69 73 74 78 81  d2 1d 99 c7 61 9d 03 00   ayListx. ....a...
0070  01 49 00 04 73 69 7a 65  78 70 00 00 00 02 77 04   .I..size xp....w.
0080  00 00 00 02 74 00 0e 2d  2d 75 6e 70 72 69 76 69   ....t..- -unprivi
0090  6c 65 67 65 64 74 00 09  31 32 37 2e 30 2e 30 2e   legedt.. 127.0.0.
00a0  31 78 73 72 00 1a 63 6f  6d 6d 6f 6e 2e 70 61 63   1xsr..co mmon.pac
00b0  6b 65 74 2e 43 6c 69 65  6e 74 50 61 63 6b 65 74   ket.Clie ntPacket
00c0  00 00 00 00 00 00 00 01  02 00 02 4c 00 04 6e 61   ........ ...L..na
00d0  6d 65 71 00 7e 00 01 4c  00 07 70 61 79 6c 6f 61   meq.~..L ..payloa
00e0  64 74 00 12 4c 6a 61 76  61 2f 6c 61 6e 67 2f 4f   dt..Ljav a/lang/O
00f0  62 6a 65 63 74 3b 78 70  74 00 04 44 4f 4e 45 70   bject;xp t..DONEp
```

Figure 14. Communication between SE and EN

Highlighted in green are the relevant strings for the desired attack. When compared to **ScanSingleHost.txt**, it is easy to pick out the elements of the command in the packet.

Once the traffic format was identified, it needed only to be replaced by some other malicious text. Ettercap had this capability, forgoing the need to find a supporting application. Ettercap provided for the use of what it called "filters." These filters performed arbitrary tasks on each packet passing through ettercap before it was forwarded. In this case, the task was somewhat simple: identify packets with the string **NMAP**, and in such cases, replace instances of **--unprivileged** and **127.0.0.1** with new arguments demonstrating the weakness of the communications method used by MAST.

Writing a filter for ettercap was similar to writing a program in a general purpose scripting language like Python or Perl. Below is the source code used to create the filter for this project, which was named **hack.filter**:

```
if (ip.proto == TCP) {
    if (search(DATA.data, "NMAP")) {
        replace("127.0.0.1," "192.*.*.*");
        replace("--unprivileged," "--packet-trace");
        msg("Got'em!\n");
    }
}
```

Line (2) searched for any TCP segment that contained ASCII text containing **NMAP**. Any other packet was simply forwarded. Should it find **NMAP**, line (3) would replace any instance of **127.0.0.1** with **192.*.*.*** and line (4) would replace any instance of **--unprivileged** with **--packet-trace**. While not necessary for the attack, line (5) notified the attacker whenever such actions were executed in order to provide visibility into the action.

The specific goal of this script was to change the console/terminal command:

```
$ nmap --unprivileged 127.0.0.1
```

to

```
$ nmap --packet-trace 192.*.*.*
```

The former conducted a simple port scan of the EN (the node which received the command). This was an innocuous port scan that would only be visible to or affect the EN itself. The latter command would perform a verbose port scan of the entire **192.0.0.0/8** network!

3. Execution and Results

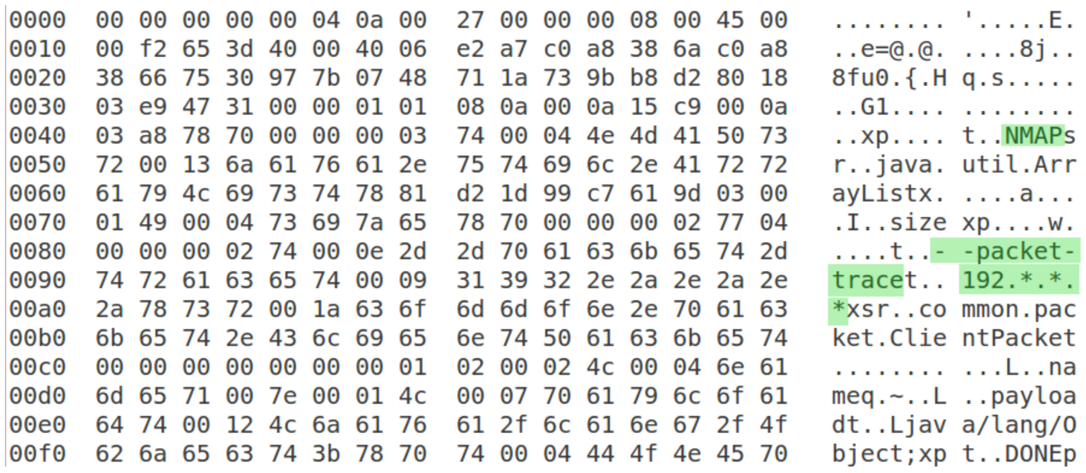
Once the ettercap filter was written, it was compiled into a binary filter file using the following command:

```
etterfilter hack.filter -o hack.ef
```

Upon compilation, the filter could be used in conjunction with any other ettercap MitM attack. In this case, it was combined with the same ARP MitM attacked described above. The resultant command was:

```
sudo ettercap -T -q -F hack.ef -M ARP /192.168.56.106// /192.168.56.102//
```

The next step was to ensure all of the VMs were running and that Wireshark was capturing traffic across **vboxnet0**. After issuing the command **start** in the SE, and applying the same Wireshark filter from above, two packets were visible. The first packet was the original (and correct NMAP) command packet sent by the SE, destined for the EN – but stopped for processing by Mallory. The second packet was the modified version sent to the EN by Mallory. Note that the original packet was not received by the intended EN. Figure 15 highlights the relevant data in the outbound packet.



```
0000   00 00 00 00 00 04 0a 00  27 00 00 00 08 00 45 00   ........ '.....E.
0010   00 f2 65 3d 40 00 40 06  e2 a7 c0 a8 38 6a c0 a8   ..e=@.@. ....8j..
0020   38 66 75 30 97 7b 07 48  71 1a 73 9b b8 d2 80 18   8fu0.{.H q.s.....
0030   03 e9 47 31 00 00 01 01  08 0a 00 0a 15 c9 00 0a   ..G1.... ........
0040   03 a8 78 70 00 00 00 03  74 00 04 4e 4d 41 50 73   ..xp.... t..NMAPs
0050   72 00 13 6a 61 76 61 2e  75 74 69 6c 2e 41 72 72   r..java. util.Arr
0060   61 79 4c 69 73 74 78 81  d2 1d 99 c7 61 9d 03 00   ayListx. ....a...
0070   01 49 00 04 73 69 7a 65  78 70 00 00 00 02 77 04   .I..size xp....w.
0080   00 00 00 02 74 00 0e 2d  2d 70 61 63 6b 65 74 2d   ....t..- -packet-
0090   74 72 61 63 65 74 00 09  31 39 32 2e 2a 2e 2a 2e   tracet.. 192.*.*.
00a0   2a 78 73 72 00 1a 63 6f  6d 6d 6f 6e 2e 70 61 63   *xsr..co mmon.pac
00b0   6b 65 74 2e 43 6c 69 65  6e 74 50 61 63 6b 65 74   ket.Clie ntPacket
00c0   00 00 00 00 00 00 00 01  02 00 02 4c 00 04 6e 61   ........ ...L..na
00d0   6d 65 71 00 7e 00 01 4c  00 07 70 61 79 6c 6f 61   meq.~..L ..payloa
00e0   64 74 00 12 4c 6a 61 76  61 2f 6c 61 6e 67 2f 4f   dt..Ljav a/lang/O
00f0   62 6a 65 63 74 3b 78 70  74 00 04 44 4f 4e 45 70   bject;xp t..DONEp
```

Figure 15.   Compromised Packet

Upon receipt, and without having any indication that the instruction was malicious, the EN began an intensive scan of the **192.0.0.0/8** network. The EN's use of system resources spiked significantly as it struggled to process the vast scan.

The selection of **192.0.0.0/8** was specific to the testing environment but could easily be tailored to more specific (or broader) needs. Furthermore, this attack could easily be extended to conduct this attack on *all* ENs on a given broadcast domain. Depending on the network, ettercap is also capable of attacking any or all hosts on a given subnet! Clearly, this attack demonstrates a severe vulnerability with the current implementation.

The previous exploitation was an attack against communication in the clear and demonstrates a vulnerability of MAST without encrypted communication. The SVN 347 version of MAST could not be tested for vulnerabilities with regard to file transferring between the SG and the SE because at the time of cloning the SVN 347 version that functionality did not exist. The lack of this feature did not allow for testing and therefore had to be added with the proposed solution. The proposed solution addressed digitally signing files that were transferred between the SG and SE.

C. PROPOSED SOLUTION TESTING

The implementation of the proposed solution is provided in the Appendix. The initial setup of the improved MAST is the same as the SVN 347 version and thus will not be covered again. The connections between the SG, SE, and EN are all secured by an

SSL connection and therefore encrypted and secure between all nodes of the MAST architecture. The same tests are conducted against the proposed solution as well as additional testing for the file transfer and digital signing of the files that are transferred.

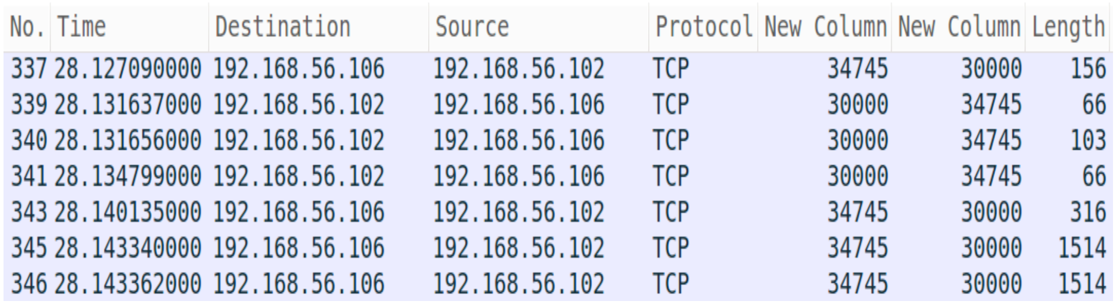
1. Re-attacking Communication

As referenced before, the communication between the SE and EN is vulnerable to a MitM attack. With the inclusion of SSL for the communication between the SE and EN, the MitM attack should be removed from the list of vulnerabilities. This is demonstrated in conducting the same MitM attack as before and finding different results. In Figure 16 the packet capture is shown for the MitM attack for the SE and EN however now the connection is using SSL and thus the packet data is represented as unintelligible ASCII non-printing characters, represented by the printable ASCII character 0x2E (a period: ".").

```
0000  00 00 00 00 00 02 0a 00  27 00 00 00 08 00 45 00   ........ '.....E.
0010  00 a3 7b 89 40 00 40 06  cc aa c0 a8 38 66 c0 a8   ..{.@.@. ....8f..
0020  38 6a d8 30 75 30 94 94  a4 4d 1a e0 c9 ed 80 19   8j.0u0.. .M......
0030  03 d4 61 7e 00 00 01 01  08 0a 00 11 ea c2 00 11   ..a~.... ........
0040  ea b3 17 03 01 00 20 04  cf 12 2d fb 41 d7 49 27   ...... . ..-.A.I'
0050  de eb 93 74 47 f7 5a 89  49 f9 b5 44 03 63 61 f0   ...tG.Z. I..D.ca.
0060  7e c6 cf 82 d8 67 f3 17  03 01 00 20 8c 6e f8 99   ~....g.. ... .n..
0070  04 8d 84 f8 d4 16 5a 77  a3 d3 95 b5 91 78 f3 14   ......Zw .....x..
0080  a8 2e d4 b8 55 8c 24 f5  10 e1 23 fb 15 03 01 00   ....U.$. ..#.....
0090  20 dd 13 b7 3a 1a 78 02  74 9d ee 19 cb 02 9f 31    ...:.x. t......1
00a0  5f 9b f3 e8 ac 6a 19 4d  df 03 ba 26 62 63 cf d4   _....j.M ...&bc..
00b0  b2                                                 .
```

Figure 16. SSL Packet Capture

Not only is the data of each packet sent between the components of MAST encrypted, but the use of SSL is also hidden from Mallory as seen in Figure 17. The packets are only seen as being sent using the TCP protocol and not one of the SSL protocols.

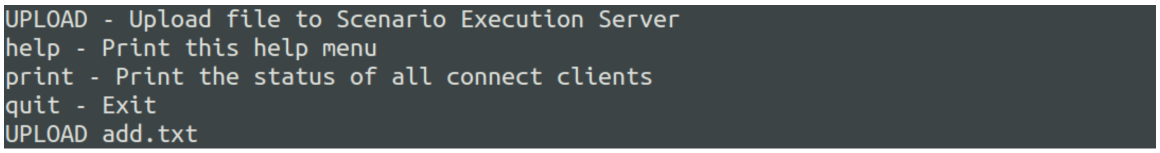


No.	Time	Destination	Source	Protocol	New Column	New Column	Length
337	28.127090000	192.168.56.106	192.168.56.102	TCP	34745	30000	156
339	28.131637000	192.168.56.102	192.168.56.106	TCP	30000	34745	66
340	28.131656000	192.168.56.102	192.168.56.106	TCP	30000	34745	103
341	28.134799000	192.168.56.102	192.168.56.106	TCP	30000	34745	66
343	28.140135000	192.168.56.106	192.168.56.102	TCP	34745	30000	316
345	28.143340000	192.168.56.106	192.168.56.102	TCP	34745	30000	1514
346	28.143362000	192.168.56.106	192.168.56.102	TCP	34745	30000	1514

Figure 17. SSL Encapsulated in TCP Session

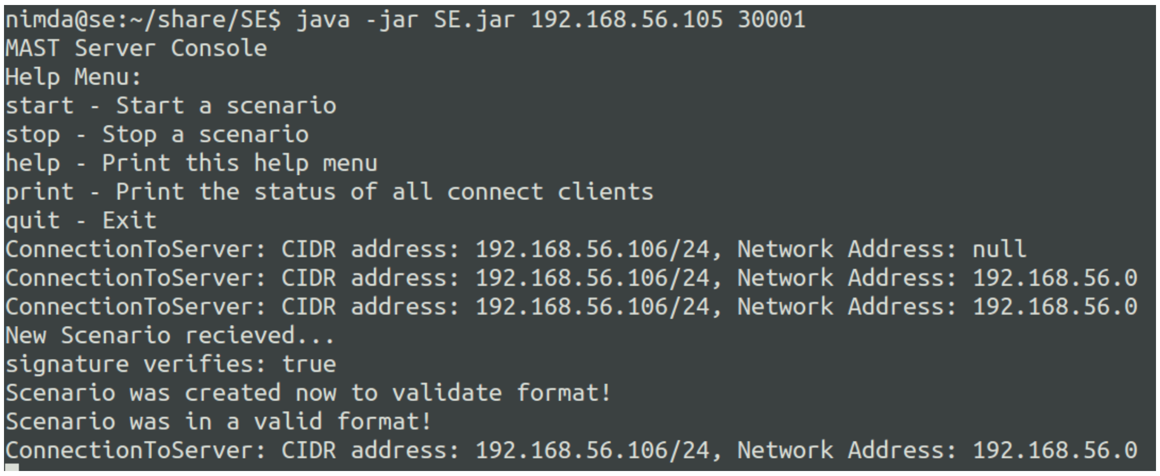
2. Attacking File Transfer

MAST had no file transfer functionality in SVN 347; this functionality is added with the addition of the proposed solution. The SG now accesses a scenario from the SG scenario folder and sends it along with the signature to the SE. The SE verifies the files as described in Chapter II. The successful transfer of a file is depicted in the SG output, in Figure 18, for sending the file while the SE output, in Figure 19, is the reception and successful verification and validation of the file.



```
UPLOAD - Upload file to Scenario Execution Server
help - Print this help menu
print - Print the status of all connect clients
quit - Exit
UPLOAD add.txt
```

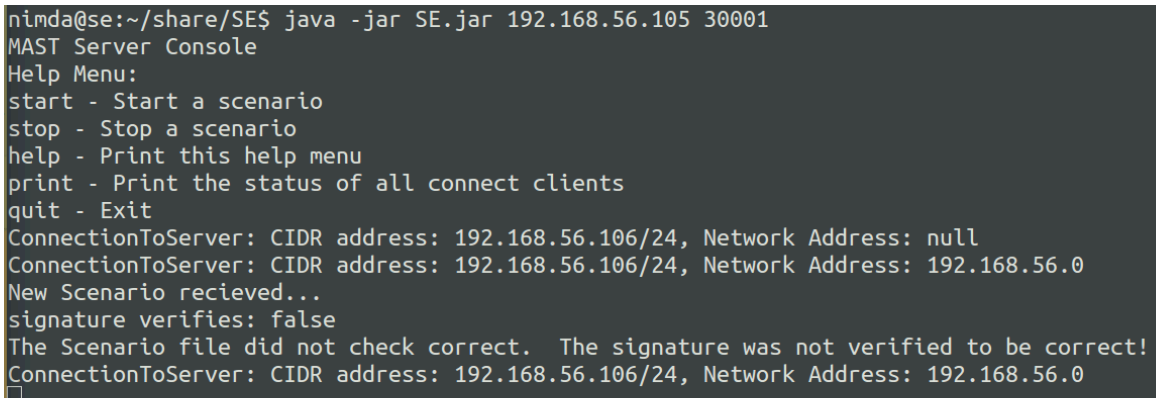
Figure 18. SG File Upload



```
nimda@se:~/share/SE$ java -jar SE.jar 192.168.56.105 30001
MAST Server Console
Help Menu:
start - Start a scenario
stop - Stop a scenario
help - Print this help menu
print - Print the status of all connect clients
quit - Exit
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: null
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: 192.168.56.0
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: 192.168.56.0
New Scenario recieved...
signature verifies: true
Scenario was created now to validate format!
Scenario was in a valid format!
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: 192.168.56.0
```

Figure 19. SE File Check and Accept

Exploitation of the file transfer via a MitM attack is no longer possible due to the proposed solution's utilization of SSL, as discussed before. However, if an insider manipulates a scenario on the SG and sends it to the SE, the SE needs to detect and appropriately respond to the file. In the proposed solution, the method by which it detects the unauthorized modification of the file is through the digital signature. Figure 20 shows how the SE will display to the user that the file does not verify with its signature.



```
nimda@se:~/share/SE$ java -jar SE.jar 192.168.56.105 30001
MAST Server Console
Help Menu:
start - Start a scenario
stop - Stop a scenario
help - Print this help menu
print - Print the status of all connect clients
quit - Exit
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: null
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: 192.168.56.0
New Scenario recieved...
signature verifies: false
The Scenario file did not check correct.  The signature was not verified to be correct!
ConnectionToServer: CIDR address: 192.168.56.106/24, Network Address: 192.168.56.0
```

Figure 20. File Fails to Verify with Signature

D. SUMMARY

While testing has not been all-inclusive for the issues addressed in Chapter III, it shows that there were vulnerabilities in MAST that could cause real world problems. The ability for an attacker to manipulate commands between the SG, SE and EN in such a manner could have far-reaching and devastating effects. If the file transfer had existed but the communication was unsecure or the file was not verified before use, an arbitrary scenario could have been uploaded to an SE.

The proposed solution addressed these issues with the addition of SSL and the digital signature of file transfers. This solution is not a panacea for the vulnerabilities that are in MAST but this testing shows that the proposed solution provides several much-needed layers of security. The demonstrated attacks were no longer successful after the utilization of SSL and digital signatures.

V. CONCLUSION AND FUTURE WORK

This chapter presents a conclusion of findings of the proposed solution as well as identifies areas of future work for MAST.

A. CONCLUSION OF FINDINGS

MAST provides an important tool for training and evaluating network administrators, as well as users, of the Navy and more importantly the DOD in general. A significant hurdle to MAST's adoption by the DOD community is expected to be the certification and accreditation issues raised in [1]. MAST has been incrementally developed by many students at NPS over the last few years. While the current implementation of MAST demonstrates the value of its approach, from a software engineering perspective it lacks in the rigor and structure of commercial products. To be deployment ready, MAST should be re-implemented with proper software methodologies and Object Oriented Programming (OOP) practices, keeping in mind the C&A issues as identified in [1]. As discussed in Chapter III, the current implementation of MAST could prevent future developers from improving upon MAST without inescapably rewriting substantial portions of the core code.

This thesis corrects seven key issues with MAST and addresses numerous issues related to the implementation, which can be seen in the Appendix. The proposed solutions follow OOP best practices when possible, deviating therefrom only when the current architecture and implementation made it impossible. The issues addressed are key to the progression of MAST through another round of DOD RMF, implementing vital layers of security. The solution presented herein has implemented encrypted communications between all components of MAST, to include communication between the SGs and SEs as well as the SEs and ENs; digitally signed file transfers between the SGs and SEs. This work has significantly hardened MAST with respect to security by ensuring it uses strong encryption, at least one-directional authentication, and limited guarantees of data integrity; thus it enhances MAST's confidentiality, integrity, availability, as well as authentication and non-repudiation capabilities.

B. FUTURE WORKS

While reviewing nearly all of MAST's source code and architecture, multiple areas of future work were found. These areas include: continued work on vulnerabilities addressed in [1] and a complete rebuild of MAST using OOP best practices.

1. Vulnerabilities from DOD RMF

This thesis addressed seven of the 77 vulnerabilities found in [1]. While most of the remaining vulnerabilities lie within the development of MAST, there are still vulnerabilities to be addressed from deployment through documentation of MAST. All of the remaining vulnerabilities should be addressed before MAST is subjected to another DOD RMF. A cursory review of the remaining vulnerabilities shows eight outstanding "High" threats to MAST in the development portion of the DOD RMF, yet solutions could be as easy as assessing the feasibility of such vulnerabilities. Impractical exploits which aim to take advantage of theoretical vulnerabilities pose less of a threat than those with are practical to exploit, perhaps even no threat at all.

2. MAST Rebuild Using OOP

The purpose and general design of MAST are well thought-out; however, as with many research projects that involve students developing the system over a number of years, MAST did not follow a product grade software engineering discipline. For many reasons, completely re-implementing MAST's functionality would be both safer and easier than attempting to rectify the noted deficiencies in [1]. Most, if not all, from our perspective, concepts of OOP have not been fully enforced in MAST's development; this lends itself to significant complications when trying to improve upon MAST and when formally analyzing it.

To be deployment ready on DOD networks, MAST should be developed by experienced software and product engineers using appropriate software methodologies. The key stakeholders should be identified and involved from the outset with clearly identified requirements and documentation maintained to the greatest degree possible to facilitate future development. Such action would address many, if not most, of the issues remaining to be addressed from [1]. Once a coherent architecture is formalized, any number of software development methodologies, such as waterfall or agile, and OOP practices should be able to generate a robust, functional, and extensible system with the potential to become a game-changer for cyber security training and readiness.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX. PROPOSED SOLUTION

The proposed solutions to the security issues addressed by this thesis are included in this appendix. These do not represent all of the source code of MAST just the portions that were edited or added to the program to achieve the hardening of MAST.

A. DIGITALLY SIGNED FILE TRANSFER

Uploading the file and the signature via a ServerCommand function is added to the ServerCommand class of the SG package also a new class is added to the common package; this class is called SignedFile. SignedFile contains the file name, the file content, and the signature of the file. This allows for MAST to transfer a file and the associated signature from the SG to the SE.

```
public class SignedFile {

        private File                    file;

        private byte[]                  content;

        private byte[]                  signature;

        public SignedFile(File file, byte[] content, byte[] signature) {

                this.file = file;

                this.content = content;

                this.signature = signature;

        }

        public File getFile() {

                return file;

        }

        public byte[] getContent() {
```

```
		return content;
	}
	public byte[] getSignature() {
		return signature;
	}
}
public class ServerCommand extends Command {
	private static final String	SCENARIO_PATH	= "SG_scenario";
	private static final String	SIGNATURE_PATH= "ScenarioSign";
	private static final String	SIGNATURE_EXT	= ".sig";
	private SignedFile			signedFile;
	private int				intPayload;

	public ServerCommand(String name) {

		super(name);
		intPayload = 0;
	}
	public ServerCommand(String name, int intPayload) {

		super(name);
		this.intPayload = intPayload;
	}
	public ServerCommand(String name, String fileName) {
			super(name);
			String filePath = SCENARIO_PATH + File.separator + fileName;
			String signPath = SIGNATURE_PATH + File.separator +
fileName + SIGNATURE_EXT;
			File scenarioFile = new File(filePath);
			FileInputStream fileInputStream = null;
			byte[] content = new byte[(int) scenarioFile.length()];
			byte[] signature = new byte[384];
			FileInputStream sigInputStream = null;
			try {
				fileInputStream = new FileInputStream(scenarioFile);
				for (int i = 0; i < content.length; i++) {
					content[i] = (byte) fileInputStream.read();
				}
				fileInputStream.close();
				sigInputStream = new FileInputStream(signPath);
				for (int i = 0; i < signature.length; i++) {
					signature[i] = (byte) sigInputStream.read();
				}
				sigInputStream.close();
			} catch (IOException e) {
				e.printStackTrace();
			}
```

```
                this.signedFile = new SignedFile(scenarioFile, content, signature);
        }
```

The SE must determine that the client packet contains a file that has to be verified and determine if the file can be written to disk or not based on the verification. This is achieved in the ServerCommunicator of the SE by checking the name of the client packet.

```
public class ServerCommunicator implements Runnable {
        private Logger                  myLogger;
        private final static int        RETRY_TIMES             = 2;
        private final static int        RETRY_DELAY             = 15000;
        private final static int        LOOP_DELAY              = 1000;
        private final static String     PUBLIC_KEY              =   "PubKeys"
+ File.separator + "public.key";
        private static final String     SCENARIO_PATH   =       "scenario"      +
File.separator;
        private Model                           model;
        private ScenarioEngine                  scenarioEngine;
        private boolean                         running;
        private SEServer                        seServer;
        private Connection                      connection;


else if ("UPLOAD".equalsIgnoreCase(clientPacket.getName())) {
        System.out.println("New Scenario recieved...");
        SignedFile signedFile = ((SignedFile) clientPacket.getPayload());
        FileInputStream keyfis = new FileInputStream(PUBLIC_KEY);
```

```
                byte[] encKey = new byte[keyfis.available()];
                keyfis.read(encKey);
                keyfis.close();
                X509EncodedKeySpec pubKeySpec = new
        X509EncodedKeySpec(encKey);
                KeyFactory keyFactory = null;

                try {
                                keyFactory = KeyFactory.getInstance("EC");
            } catch (NoSuchAlgorithmException e) {
                System.out.println("Inside Key Factory for Server Communicator: No
        Such Algorithm Exception has been Thrown");
                                        }
            PublicKey pubKey = keyFactory.generatePublic(pubKeySpec);
            byte[] sentSignature = signedFile.getSignature();
            Signature sig = Signature.getInstance("SHA512withECDSA");
            sig.initVerify(pubKey);
            sig.update(signedFile.getContent());
            boolean verify = sig.verify(sentSignature);
            if (verify) {
            System.out.println("signature verifies: " + verify);
            File scenarioFile = signedFile.getFile();
            String uploadedScenario = SCENARIO_PATH
            scenarioFile.getName().toString();
```

```
FileOutputStream output = new FileOutputStream(uploadedScenario);

output.write(signedFile.getContent());

output.close();

System.out.println("Scenario was created now to validate format!");

ScenarioParser scenarioParser = new ScenarioParser(uploadedScenario);

File scenarioSEFile = new File (uploadedScenario);

Scenario scenario = new Scenario(scenarioSEFile);


scenarioParser.parseScenario(scenario);

System.out.println("Scenario was in a valid format!");

} else {

System.out.println("signature verifies: " + verify);

System.out.println("The Scenario file did not check correct.  The signature was
not verified to be correct!");

disconnectFromServer();

                    }

            }
```

B. SECURE SOCKET LAYER COMMUNICATION

Any time MAST communicates as a server it uses the SpawnCommunicator class in the Common package. SpawnCommunicator now uses SSL to create a socket for the communication and then once the socket is created it is then hardened to remove potential vulnerabilities. The hardening of the socket removes protocols that have known vulnverabilities.

```
public class SpawnCommunicator<T extends Communicator> implements
Runnable {
	private final static String[]	PROHIBITED_PROTOS			=	{
"SSLv2Hello", "SSLv3", "TLSv1.1"};

		private final static String[]	PROHIBITED_SUITE_COMPONENTS
	= { "anon", "DSA", "RC4", "MD5","DES", "NULL"};

		private final static boolean	EXACT = true;

		private int					listenPort;

		private int					maxClients;

		private SSLServerSocket			sslServerSocket;

		private boolean				isListening;

		private ExecutorService			pool;

		private Model					model;

		private Executable				server;


		private Class<T>				classTemplate;


		public SpawnCommunicator(int listenPort, int maxClients, Model model,
Executable server, Class<T> classTemplate) {

			this.listenPort = listenPort;

			this.maxClients = maxClients;

			this.model = model;

			this.server = server;

			this.classTemplate = classTemplate;

			this.pool = Executors.newCachedThreadPool();

			this.isListening = true;
```

```
        }
        public void run() {
        try {
        SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory)
    SSLServerSocketFactory.getDefault();
        sslServerSocket = (SSLServerSocket)
    sslServerSocketFactory.createServerSocket(listenPort, maxClients);
        hardenSSLServerSocket(sslServerSocket);
        while (this.isListening) {
        SSLSocket sslSocket = (SSLSocket) sslServerSocket.accept();
        Constructor<T> constructor =
    classTemplate.getDeclaredConstructor(SSLSocket.class, Model.class);
        Communicator communicator = constructor.newInstance(sslSocket, model);

        pool.submit(communicator);
                }
        } catch (SocketException e) {
                                e.printStackTrace();
                                shutdown(0);
        } catch (IOException e) {
                                e.printStackTrace();
                                shutdown(0);
        } catch (SecurityException e) {
                        shutdown(0);
        } catch (IllegalAccessException e) {
                        e.printStackTrace();
                        shutdown(0);
        } catch (IllegalArgumentException e) {
                        e.printStackTrace();
                        shutdown(0);
        }     catch     (InvocationTargetException     |     NoSuchMethodException     |
    InstantiationException e) {
                        e.printStackTrace();
                        shutdown(0);
            }
        }

        public void shutdown(int runLevel) {
            isListening = false;
            try {
                        sslServerSocket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                    pool.shutdown();
                    server.shutdown(runLevel + 1);
            }
```

```
                private void hardenSSLServerSocket(SSLServerSocket sslServerSocket)
{
        String[] defaultProtos = sslServerSocket.getSupportedProtocols();
        String[]        allowedProtos       =       removeInstancesOf(defaultProtos,
PROHIBITED_PROTOS, EXACT);
        sslServerSocket.setEnabledProtocols(allowedProtos);
        String[] defaultSuites = sslServerSocket.getSupportedCipherSuites();
        String[]        allowedSuites       =       removeInstancesOf(defaultSuites,
PROHIBITED_SUITE_COMPONENTS, !EXACT);
        sslServerSocket.setEnabledCipherSuites(allowedSuites);
                }
        private String[] removeInstancesOf(String[] originals, String[] prohibited_items,
                                boolean exactness) {
                ArrayList<String> resultsList = new ArrayList<String>();
                for (String original : originals) {
                        boolean prohibited = false;
                        for (String prohibited_item : prohibited_items) {
                        if ((exactness != EXACT && original.contains(prohibited_item))
                                ||      (exactness      ==      EXACT       &&
original.equals(prohibited_item))) {
                        prohibited = true;
                        break;
                }
        }
                        if (!prohibited) {
```

```
                        resultsList.add(original);
                }
        }
                return resultsList.toArray(new String[resultsList.size()]);
                }
        }
```

MAST also needs to communicate as a client and thus uses the ConnectionToServer class in the common package. SSL has been added to the class so that any time there is a client communicating with a server it will use SSL.

```
        public class ConnectionToServer {
                private final static int        SOCKET_TIMEOUT          = 1000;
                private Connection                      connection;
                private boolean                         isConnected;
                public ConnectionToServer(Connection connection) {
                        this.connection = connection;
                        this.isConnected = false;
                        initConnection();
                }
                private void initConnection() {
                        int retryCount = 0;
                        while (retryCount < connection.getRetryTimes()) {
                                try {
                                        SSLSocketFactory        sslSocketFactory        =
(SSLSocketFactory) SSLSocketFactory.getDefault();
```

```
					SSLSocket sslSocket = (SSLSocket) sslSocketFactory.createSocket();
					sslSocket.bind(null);
					InetSocketAddress inetSocketAddress = new InetSocketAddress(connection.getServerAddress().getHostAddress(),connection.getServerPort());
					sslSocket.connect(inetSocketAddress, SOCKET_TIMEOUT);
					connection.setSSLSocket(sslSocket);
					connection.setOut(new ObjectOutputStream(connection.getSSLSocket().getOutputStream()));
					connection.setIn(new ObjectInputStream(connection.getSSLSocket().getInputStream()));
					connection.setLocalAddress((Inet4Address) connection.getSSLSocket().getLocalAddress());
					NetworkInterface networkInterface = NetworkInterface.getByInetAddress(connection.getLocalAddress());
	networkInterface.getInterfaceAddresses().indexOf(connection.getLocalAddress());
					for (InterfaceAddress ifaceAddr : networkInterface.getInterfaceAddresses()) {
	If (ifaceAddr.getAddress().equals(connection.getLocalAddress())) {
		String prefix = String.valueOf((ifaceAddr.getNetworkPrefixLength() != 0) ? ifaceAddr.getNetworkPrefixLength() : 8);
		String CIDRAddress = connection.getLocalAddress().getHostAddress() + "/"+ prefix;
	System.out.println("ConnectionToServer: CIDR address: " + CIDRAddress
                                                            +       ",       Network
Address: " + connection.getNetworkAddress());
        SubnetUtils utils = new SubnetUtils(CIDRAddress);
        SubnetInfo info = utils.getInfo();
        connection.setNetworkAddress(info.getNetworkAddress());
                    }
                }
                isConnected = true;
                return;
                        } catch (UnknownHostException e) {
                connection.getLogger().log(Level.WARNING,"Can't       find       "       +
connection.getServerAddress());
                            try {
                    Thread.sleep(connection.getRetryDelay());
                        } catch (InterruptedException e1) {
                connection.getLogger().log(Level.WARNING,
e1.getStackTrace().toString());
                        }
                    retryCount++;
                        } catch (IOException e) {
                connection.getLogger().log(Level.WARNING,"Couldn't  get  I/O  for  "  +
connection.getServerAddress());
                            try {
                                Thread.sleep(connection.getRetryDelay());
                            } catch (InterruptedException e1) {
```

```
connection.getLogger().log(Level.WARNING, e1.getStackTrace().toString());
                    }
                            retryCount++;
                        }
                    }
                    connection.getLogger().log(Level.WARNING,
            "Tried " + connection.getRetryTimes() + " times, giving up");
                    isConnected = false;
                    return;
            }
            public boolean isConnected() {
                    return isConnected;
            }
    }
```

LIST OF REFERENCES

[1] B. J. Diana, “Malicious Activity Simulation Tool (MAST) and trust,” M.S. thesis, CS, NPS, Monterey, CA, 2015.

[2] C. Pfleeger, S. Pfleeger and J. Margulies, *Security in Computing*. Westford, MA: Prentice Hall, 2015.

[3] T. Kohno, N. Ferguson and B. Schneier, *Cryptography Engineering*. Indianapolis, IN: Wiley Pub., Inc., 2010.

[4] S. Singh, *The Code Cook*. New York: Doubleday, 1999.

[5] Data Encryption Standard, FIPS 46–3, Oct. 1999.

[6] E. Biham and A. Shamir “Differential Cryptanalysis of DES-like cryptosystems” Dept. of Appl. Math., The Weizmann Institute of Science, July 19, 1990.

[7] Advanced Encryption Standard, FIPS 197, Nov. 2001

[8] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, IEEE Standard 802.11, 1999.

[9] The AES-CBC cipher algorithm and its use with IPsec, RFC 3602, Sep. 2003.

[10] Rivest, R. L., A. Shamir and L. M. Adleman, A Method for Obtaining Digital Signatures and Public Key Cryptosystems, Communications of the ACM, Volume 21, Number 2, February 1978, (pp. 120–126).

[11] Introduction to higher mathematics. (n.d.). P. Keef and D. Guichard. [Online]. Available: http://www.whitman.edu/mathematics/higher_math_online/section03.08.html. Accessed on Mar. 24, 2015.

[12] Secure hashing. (2015, August 6). NIST. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html

[13] Privacy Enhancement for Internet Electronic Mail, RFC 1422, Feb. 1993

[14] Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile, RFC 3280, Apr. 2002.

[15] Java Security Overview. (n.d). [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/security/overview/jsoverview.html. Accessed on Mar. 24, 2015.

[16] List all provider and its algorithm. (n.d.). [Online]. Available: http://www.java2s.com/Code/Java/Security/ListAllProviderAndItsAlgorithms.htm. Accessed on Mar. 24, 2015.

[17] Security Requirements for Cryptographic Modules, FIPS 140–2, May 2001.

[18] The Transport Layer Security (TLS) Protocol Version 1.2, RFC 5246, Aug. 2008.

[19] Secure Socket Layer (SSL) Protocol Version 3.0, RFC 6101, Aug. 2011.

[20] The TLS Protocol, RFC 2246, Jan. 1999.

[21] The Transport Layer Protocol (TLS) Version 1.1, RFC 4346, Apr. 2006 .

[22] Guidelines for the Selection , Configuration, and Use of Transport Layer Security (TLS) Implementations, NIST SP 800–52rl, Apr. 2014.

[23] H. Johnson, "Web Security" in Network Security, Blekinge Institute of Technology, Sweden, 2012.

[24] Prohibiting RC4 Cipher Suites, RFC 7465, Feb. 2015.

[25] Addition of ARIA Cipher Suites to Transport Layer Security (TLS), RFC 6209, Apr. 2001.

[26] Addition of Camellia Cipher Suites to Transport Layer Security (TLS), RFC 6367, Sep. 2011.

[27] D. Adrian et al., "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice," WeakDH.org [Online]. Available: https://weakdh.org/imperfect-forward-secrecy.pdf Accessed: Jun. 24, 2015.

[28] M. Vanhoef and F. Pissans, "All Your Biases Belong To Us," in USINEX 2015 © USINEX 2015.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California